Volume 5

Media in Context

Enabling Technology Modules: Final Version

Responsible editor(s): Patrick Aichroth, Johanna Björklund,
Emanuel Berndl, Thomas Kurz, Thomas Köllmer

Volume 5

Enabling Technology Modules: Final Version

Patrick Aichroth, Johanna Björklund, Emanuel Berndl, Thomas Kurz, Thomas Köllmer

About the project: MICO is a research project project partially funded by the European Commission 7th Framework Programme (grant agreement no: 610480). It aims to provide cross-media analysis solutions for online multimedia producers. MICO will develop models, standards and software tools to jointly analyse, query and retrieve information out of connected and related media objects (text, image, audio, video, office documents) to provide better information extraction results for more relevant search and information discovery.

Abstract: This Technical Report summarizes the state of the art in cross-media analysis, metadata publishing, querying and recommendations. It is a joint outcome of work packages WP2, WP3, WP4 and WP5, and serves as entry point and reference for technologies that are relevant to the MICO framework and the two MICO use cases.

Projekt Coordinator: John Pereira BA
Publisher: Salzburg Research Forschungsgesellschaft mbH, Salzburg, Austria
Editor of the series: Thomas Kurz | **Contact:** thomas.kurz@salzburgresearch.at
Issue: December, 2016 | **Grafik Design:** Daniela Gnad
ISBN 978-3-902448-47-7

Images are taken from the Zooniverse crowdsourcing project Plankton Portal that will apply MICO technology to better analyse the multimedia content. https://www.zooniverse.org

Online: A digital version of the handbook can be freely downloaded at http://www.mico-project.eu/technical-reports/

MICO (Media in Context) is a research project partially funded by the European Commission 7th Framework Programme (grant agreement no: 610480).

Responsible editors

Patrick Aichroth is working for Fraunhofer IDMT and focusing on user-centric, incentive-oriented solutions to the“copyright dilemma”, content distribution and media security. Since 2006, he is head of the media distribution and security research group. Within MICO, he is coordinating FHG activities, and is involved especially in requirements methodology and gathering, related media extractor planning, and system design and implementation aspects related to broker, media extraction and storage, and security.

Johanna Björklund is senior lecturer at the Department of Computing Science at Umeå University, and founder and COO of CodeMill, an IT-consultancy company specializing in media asset management (MAM). Her scientific work focuses on structured methods for text classification. In MICO she is leading the activities around a multi-lingual speech-to-text component.

Emanuel Berndl is a PhD Student at the University of Passau. He is mainly working on Semantic Web, Software Engineering and modern Web Technologies. In the MICO project he is contributing to the multimedia annotation model, which also builds the core of his research interests.

Thomas Kurz is Researcher at the Knowledge and Media Technologies group of Salzburg Research. His research interests are Semantic Web technologies in combination with multimedia, human-computer interaction regarding to RDF metadata, and Semantic Search. In Mico he focuses on semantic multimedia retrieval and coordinates the overall scientific work.

Thomas Köllmer is working for Fraunhofer IDMT within the media distribution and security research group, and as a researcher inside the media technology department of the Ilmenau University of Technology. He is working on metadata modelling, the objective assessment of recommendation quality and the development of novel recommendation methods. In MICO, he focusses on developing and integrating methods for cross media recommendation.

Responsible authors

Many different authors from all partners have contributed to this document. The individual authors in alphabetic order are:

- **Patrick Aichroth** (FhG),
- **Johanna Björklund** (UMU),
- **Emanuel Berndl** (University of Passau, UP),
- **Alex Bowyer** (University of Oxford),
- **Luca Cuccovillo** (FhG),
- **Thomas Kurz** (Salzburg Research Forschungsgesellschaft mbH, SRFG),
- **Thomas Köllmer** (FhG),
- **Marcel Sieland** (FhG),
- **Christian Weigel** (FhG)**,**
- **Thomas Weißgerber** (UP)

MICO – Volume

Volume 1

State of the Art in Cross-Media Analysis, Metadata Publishing, Querying and Recommendations

Patrick Aichroth, Johanna Björklund, Florian Stegmaier, Thomas Kurz, Grant Miller

ISBN 978-3-902448-43-9

Volume 2

Specifications and Models for Cross-Media Extraction, Metadata Publishing, Querying and Recommendations: Version I

Patrick Aichroth, Henrik Björklund, Johanna Björklund, Kai Schlegel, Thomas Kurz, Grant Miller

ISBN 978-3-902448-44-6

Volume 3

Enabling Technology Modules: Version I

Patrick Aichroth, Henrik Björklund, Johanna Björklund, Kai Schlegel, Thomas Kurz, Antonio Perez

ISBN 978-3-902448-45-3

Volume 4

Specifications and Models for Cross-Media Extraction, Metadata Publishing, Querying and Recommendations: Version II

Patrick Aichroth, Johanna Björklund, Kai Schlegel, Thomas Kurz, Thomas Köllmer

ISBN 978-3-902448-46-0

Volume 5

Enabling Technology Modules: Final Version

Patrick Aichroth, Johanna Björklund, Emanuel Berndl, Thomas Kurz, Thomas Köllmer

ISBN 978-3-902448-47-7

Contents

List of Figures

List of Tables

1 Executive Summary

This document provides the final description of the MICO technology enablers from WP2, WP3, WP4 and WP5, including several system domain updates (model / persistence, broker, querying, and recommendation framework) since Year 2.

Section 2 provides the relevant updates regarding MICO extractors and their implementation status with respect to the specifications of (Mico Technical Report V4) related to linguistic analysis, including diarization, ASR, text and competence classification, language detection, and NER. It also includes a description of two new extractors for animal vs. blank image distinction in Year 3. Finally, Section 2 describes how the MICO extractor API evolved since the MICO Technical Report Volume 3, and how to implement a MICO extractor.

Section 3 describes the MICO broker v3 for extractor orchestration, which represents a major update over v1 provided with report V3, and also over v2 that was provided earlier in Year 3. The section outlines requirements and design principles, describes the broker data model, provides a component overview, and presents the registration service and its interplay with extractors. It includes an explanation of how semi-automatic creation and workflow execution has been implemented. Section 3 also provides a description of workflow management tools that were not planned for Year 3, and are not part of and not needed to use the platform, but nonetheless improve usability.

Section 4 provides an updated description of Anno4j, the Object-to-RDF ORM mapping for MICO that allows the creation of MICO or generic W3C Web Annotations with plain Java objects. Anno4j was originally introduced in report V3, and has been significantly extended since then.

Section 5 updates the description of the query language extension SPARQL-MM to version 2.0. It then continues to describe how LDPath, a simple path-based query language, similar to XPath, can be used for simplified querying and data retrieval. LDPath is here provided as an alternative to SPARQL-MM, which was used in earlier reports. The conclusion of Section 5 presents a solution approach to the problem of Semantic Media Similarity.

Section 6 describes the MICO recommendation framework, including status and usage instructions for the related components, and a description of the recommendation API and respective demo code.

2 Enabling Technology Modules for Extractors

This section provides updates on the MICO extractors and their implementation status with respect to the specifications of report V4. Several of these concern linguistic analysis, e.g., diarization, automatic speech recognition, sentiment analysis, language detection, and named-entity recognition. We also describe how the MICO extractor API has evolved since the MICO Technical Report Volume 3, and explain how to implement a MICO extractor.

2.1 Visual Extractors Status

There have been relatively few functional updates to the video and image related extractors since the MICO Technical Report Volume 4. Implementation efforts have mostly concerned bug fixes and the adaptation to the new C++ and Java APIs (see Section 2.6.1). However, as planned in Volume 3, there have been updates to the animal detection extractor as well as the adaptation to Broker v3 in the final year of the project.

2.1.1 Object and Animal Detection – OAD (TE-202) – Update

As planned in report V4 we have extended the animal detector using DPMs (Deformable Part Models [Fel+10]) for detection. Yet some experiments showed that these hand-crafted features still have difficulties with the demanding image data set of Snapshot Serengeti. Therefore we decided to add another implementation of the animal detection service that relies on Deep Neural Networks. We chose the very recently published YOLO framework [Red+16] for implementation and trained our own model for 10 species based on the data used for HoG based and DPM training. We have also got approx. 10.000 additional annotations done by the Zooniverse community in the Snapshot Serengeti CV project that was set up during MICO (`https://www.zooniverse.org/projects/alexbfree/computer-vision-serengeti/`). Unfortunately due to time constraints this valuable data could not make it into the training data set. Yet, we will use it for a final evaluation of the detection performance and perhaps a re-training.

Due to the additional efforts for integration of a third animal detector we decided to reduce efforts on the alternative approaches for blank image detection (background model etc.). We derived interesting statistics about the Snapshot Serengeti sets and applied experimental algorithms (e.g., day/night detectors, background models) but these research results are in a too unfinished state in order to create a MICO extractor out of it. However, we are looking forward that the DNN approach for detection also helps in the blank image detection task.

2.2 Audio Extractors Status

The audio extractors include (i) diarization to segment audio data into manageable chunks, (ii) automatic speech recognition to transcribe audio data into text, and (iii) auxiliary extractors to store the transcription as XML. The work done in Year 3 of the project has focused on providing an alternative to Kaldi by wrapping the Microsoft Bing Speech service, and by improving the multilingual support by training new models for Kaldi.

2.2.1 Automatic Speech Recognition based on Kaldi (TE-214) – Update

Automatic speech recognition (ASR) takes as input an audio stream with speech and outputs a transcription as plain text. ASR is useful in itself, for example, to generate subtitles or make content indexable via

text-driven approaches. ASR is also needed as an intermediate step towards sentiment analysis, named entity recognition and topic classification. In the context of the MICO Showcases, ASR is relevant for the News Media scenario (US-18), to make videos searchable through keywords.

In the initial phase of the MICO project, we evaluated several open-source and proprietary ASR libraries. The outcome of the experiments are published on the project's webpage[1]. Given the results, we decided to base MICO's ASR capabilities on the C++ library Kaldi, as it outperformed the other open-source alternatives and was at the level of some of the commercial systems. Kaldi also has the advantages of being modern and extensible, and having a business-friendly license.

Table 1 TE-214 implementation: ASR implementation with the Kaldi library

Name	mico-extractor-speech-to-text (`http://kaldi-asr.org/`)
Original license	Apache license, version 2.0
MICO integration license	Apache license, version 2.0
External dependencies	Kaldi, Boost, ATLAS, BLAS, gfortran3
Input data	Diarization segment information (XML) and corresponding audio file (WAV).)
Output data	Time-stamped word transcriptions (XML)
RDF persistence	Persistence into the MICO data model via a separate Java extractor converting each time-stamped word into a content part.
Internal Parameters	*SAMPLING_RATE*: 8000, *MAX_SEGMENT_SIZE_SEC*: 30 (used to break too large segments down).
Additional requirements	A language model must be provided (default installed with Debian package)

Comments: ASR is still in its infancy, and requires time and effort to mature. Leading commercial systems achieve an accuracy of approx. 85% in the best possible setting, that is, with a single male speaker working from a manuscript, with no background noise and high-quality audio. Open-source systems tend to lie 10-15% lower on the same content. Under less favorable circumstances, for example a typical YouTube clip, the accuracy can easily drop to 50%. Despite these facts, the output can still be useful to search content by keywords, or to sort it by topic.

Another aspect worth mentioning is that ASR systems are very resource intensive. The language-models used to represent language typically requires a couple of gigabytes of disk, and the computations involved are CPU intensive. Future systems will likely have similar requirements, but this will be mitigated by faster processing kernels, higher parallelization, and lower costs of disk and memory. Further work with Kaldi should focus on offline transcription with better parallel capabilities. In current

[1] http://www.mico-project.eu/experiences-from-development-with-open-source-speech-recognition-libraries/

tests, the extractor can extract text from speech in about 160 per cent of real time, which gives good hope of achieving real time transcription with further optimizations and hardware improvements.

The early implementations of the Kaldi extractor suffered from being resource intensive. An effort to mitigate this was taken and the internal parameters of the Kaldi system and their influence on the quality of the transcription results was evaluated. Tests on a parameter space of five parameters with 4-8 different values was performed. This gave insight into parameter settings for different purposes (low error rate, high transcription speed, and so forth). It is advisable to keep the default parameter settings if the transcription results are needed for text analysis. However, in applications where 'perfect' transcriptions are not needed (e.g. keyword, topic detection, etc.) there are more effective settings. The extractor is therefore published together with a list of suggested parameter settings for varying trade-off levels in speed and accuracy.

2.2.2 Automatic Speech Recognition based on Microsoft's Bing Voice Recognition (TE-214) – New

Kaldi is free to use and can be trained for any language, but since the library is still comparatively new, few high-quality language models exist. To offer a broad language support, we therefore include an alternative ASR extractor based on Microsoft's cloud service Bing Voice Recognition. Bing Voice Recognition currently supports 20 languages, including Italian and Modern Standard Arabic, which are of particular interest for the MICO use cases. It also shows how existing SaaS solutions can be integrated into the MICO platform.

The service is accessed through a REST API and returns one recognition result with no partial results. A sample voice recognition request looks as follows:

```
POST /recognize? (parameters)
Host: speech.platform.bing.com
Content-Type: audio/wav; samplerate=16000
Authorization: Bearer [Base64 access_token]

(audio data)
```

The service returns a response in JSON format, the content of which depends on whether the provided audio led to a successful recognition. If it did, the response may look as follows:

```
HTTP/1.1 200 OK
Content-Length: XXX
Content-Type: application/json; charset=UTF-8
{

  "version":"3.0",
  "header":{
    "status":"success",
    "scenario":"websearch",
    "name":"Mc Dermant Autos",
    "lexical":"mac dermant autos",
    "properties":{
      "requestid":"ABDDD97E-171F-4B75-A491-A977027B0BC3"
    },
  "results":[{
    # Formatted result
    "name":"Mc Dermant Autos",
```

```
    # The text of what was spoken
    "lexical":"mac dermant autos",
    "confidence":"0.9442599",
    # Words that make up the result; a word can include a space if there
    # isn't supposed to be a pause between words when speaking them
    "tokens":[{
      # The text in the grammar that matched what was spoken for this token
      "token":"mc dermant",
      # The text of what was spoken for this token
      "lexical":"mac dermant",
      # The IPA pronunciation of this token (I made up M AC DIR MANT;
      # refer to a real IPA spec for the text of an actual pronunciation)
      "pronunciation":"M AC DIR MANT",
    },
    {
      "token":"autos",
      "lexical":"autos",
      "pronunciation":"OW TOS",
    }],
    "properties":{
      "HIGHCONF":"1"
    },
  }],
  }
}
```

A failure to recognize the content instead leads to the following reply:

```
HTTP/1.1 200 OK
Content-Length: XXX
Content-Type: application/json; charset=UTF-8

{
   "version":"3.0",
   "results":[{}],
   "header":{
     "status":"error",
     "scenario":"websearch",
     "properties":{ "requestid":"ABDDD97E-171F-4B75-A491-A977027B0BC3",
       "FALSERECO":"1"
     }
   }
}
```

Comments: When choosing a commercial alternative to Kaldi, we evaluated Microsoft's and Google's speech-recognition services. Both services supported Italian and Arabic, which is a must for our showcases. Google had the advantages of having a simple API, but the disadvantages of still being under restricted Beta access, requiring audio in FLAC format (which would involve a conversion step), the files are restricted to 10-15 seconds, and there is a maximum of 50 requests per day.

The main advantage of Microsoft's service was that it permits 5000 requests per month free of charges[2]; the main disadvantage that it has two APIs (Bing Speech and Bing Voice Recognition), and it

[2]Larger volumes are available for purchase.

Table 2 TE-214 implementation: ASR implementation with Microsoft's Bing Voice Recognition Service

Name	mico-extractor-speech-to-text-ms (`https://datamarket.azure.com/dataset/bing/speechrecognition`)
Original license	Free access, 5000 requests per month, subscription Terms of Use
MICO integration license	Apache license, version 2.0
External dependencies	Microsoft Azure Cloud platform, active Bing Speech subscription
Input data	Audio file (with sample rate 8000 or 16000)
Output data	Plain text transcription
RDF persistence	Persistence into the MICO data model converting the plain text into a content part.
External Parameters	Locale at startup (en-US default)

is hard to differentiate between them and understand what restrictions apply where. This extractor uses Bing Voice Recognition, which is the Cloud Beta version of the service. Microsoft's API also requires you to set up sessions instead of only using a subscription key, and files are restricted to 10 seconds, with a maximum of 20 requests per minute for the non-cloud API and an undisclosed limit for the Cloud API with the guideline of 1 request per second. As the extractor makes only synchronous requests this limit is not an issue.

All things considered, we decided to use Microsoft's Bing Voice Recognition. The languages and locales supported by the Bing Speech API are listed online[3] and are supported by the extractor.

2.3 Textual Extractors Status

For textual extractors, we report on the major advances since MICO Technical Report Volume .

2.3.1 OpenNLP Text Classification Extractor (TE–213) – New

This extractor classifies textual content along trained categories. The classification is done by using Apache OpenNLP's Document Categorizer (DocCat) functionality implementing a maximum entropy algorithm[4]. The extractor requires an OpenNLP Doccat model to be provided as an parameter. In additional an optional SKOS thesaurus can be provided that maps string names of categories trained for the model to concept URIs in the thesaurus. For doing so the string names of the categories are mapped with the skos:notation values defined by the parsed Thesaurus.

The extractor itself requires a plain/text asset to be present in the processed item. It will output a single fam:TopicClassification and one or more fam:TopicAnnotation for actual categories the processed

[3] `https://www.microsoft.com/cognitive-services/en-us/speech-api/documentation/overview`

[4] `https://opennlp.apache.org/documentation/1.6.0/manual/opennlp.html#opennlp.ml.maxent`

text was assigned to.

For the detailed definition of those annotation please see the Fusepool Annotation Modell[5]. The following listing shows an annotation result of a classification result processing a text with a model trained for two categories detecting competence and incompetence.

```
@prefix xsd: <http://www.w3.org/2001/XMLSchema#> .
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix dc:  <http://purl.org/dc/elements/1.1/> .
@prefix fam: <http://vocab.fusepool.info/fam#> .
@prefix mmm: <http://www.mico-project.eu/ns/mmm/2.0/schema#> .
@prefix oa:  <http://www.w3.org/ns/oa#> .
@prefix compclass: <http://www.mico-project.eu/ns/compclass/1.0/schema#> .
@prefix example:   <http://www.example.org/mico/competenceClassification#> .

test:topic-classification-1 a fam:TopicClassification ;
 fam:classification-scheme compclass:CompetenceClassificationScheme ;
 fam:extracted-from test:item1 ;
 oa:item test:topic-annotation-2 , test:topic-annotation-1 .

test:topic-annotation-2 a fam:TopicAnnotation ;
 fam:confidence "0.2823413046977258"^^xsd:double ;
 fam:extracted-from test:item1 ;
 fam:topic-label "Kompetent"@de , "Competent"@en ;
 fam:topic-reference compclass:Competent .

test:topic-annotation-1 a fam:TopicAnnotation;
 fam:confidence "0.7176586953022742"^^xsd:double ;
 fam:extracted-from test:item1 ;
 fam:topic-label "Inkompetent"@de , "Incompetent"@en ;
 fam:topic-reference compclass:Incompetent .
```

2.3.1.1 Specific comments

Performance: OpenNLP holds NLP models in memory. Using this Extractor will require sufficient memory for loading all configured models in memory. The memory required by a OpenNLP Doccat model depends on the number of categories and also on the size of the trainings set. Memory is expected to be below 100 MByte of RAM for most use cases. OpenNLP supports multi-threading what means that multiple concurrent requests can be processed on different CPU cores. Scaling depends therefore on the number and speed of available CPU cores.

Output: Like all MICO NLP extractors, this extractor creates RDF annotations according to the MICO metadata model v2 and the Fusepool Annotation Model. In particular, it creates fam:TextClassification and fam:TopicAnnotation for representing the classification and assigned categories. Note that the extractor will only create a single classification for the whole text/plain content.

[5] https://github.com/fusepoolP3/overall-architecture/blob/master/wp3/fp-anno-model/fp-anno-model.md

Table 3 The OpenNLP Text Classification Extractor implementation (TE-213)

Name	MICO OpenNLP Text Classification Extractor
Original license	Apache Software License 2.0
MICO integration license	Apache Software License 2.0
External dependencies	This Extractor is based upon the OpenNLP Doccat (Document Categorizer) functionality.
Input data	Plain text originating from written text or from speech to text transcoding
Output data	RDF
RDF persistence	Web annotation based annotations as defined by the MICO metadata model v2.0 with a single TopicClassification and possible multiple TopicAnnotation as defined by the Fusepool Annotation Modell[6]
External Parameters	none
Internal Parameters	The Extractor requires an OpenNLP Doccat model to be configured. Optionally it can be configured with a SKOS thesaurus that maps string names of the Doccat model to Concepts of the thesaurus.

2.3.2 Competence Classification (TE–213) – New

The extractor classifies parsed text along the competence for the author. This extractor uses the Open NLP Document classification extractor configured with a model and thesaurus configured for competence classification. Its main usage within MICO is to classify user comments in Serengeti Snapshots according to the competence, or skill level, of a user.

For the training of a competence classification we extracted a (denormalized) view over the Snapshot Serengeti user data with about 100,000 entries, each containing the following fields:

1. userID – this allows the user to be uniquely identified, and that user comments to be collected together.
2. number of images annotated by this user – this defines how active a user is
3. percentage [0..1] of correctly annotated images – this gives a good estimate about the expertise of the user, especially for very active users. This is less useful for users with very few annotated images. (Most Snapshot Serengeti users leave few or no comments)
4. user comment – the body of the user comment itself

To train a categorization model one needs to create a training set. Typically, this would be done by manually classifying user comments. To skip the time consuming manual annotation, the given data were analyzed using Open Refine. Based on this analysis a set of rules where established to classify comments into the following two categories

1. COMP (competent)
2. INCOMP (incompetent)

Definitions:

- ln-anno-img := ln(1+column2): logarithm over the number of annotated images
- cor-anno := column3: percentage of correctly annotated images
- q-words := A list of words indicating a question (‘¿, ‘who‘, ‘what‘, ‘why‘, ‘may‘, ‘can’t‘, ‘can not‘)

Rules:

1. GLOBAL: only comments with >= 3 tokens are considered
2. INCOMP: ‘ln-anno-img >= 2‘ and ‘ln-anno-img <= 5‘ — Comments of Users that have only some annotated Images are considered as incompetent.
3. INCOMP: ‘ln-anno-img > 5‘ and ‘cor-anno < 0.82‘ — Comments of Users 100 or more annotated images but a low percentage of correctly annotated images are considered as incompetent.
4. INCOMP: ‘ln-anno-img > 5‘ and ‘ln-anno-img <= 9‘ and ‘cor-anno >= 0.82‘ and ‘cor-anno <= 0.85‘ and contains ‘q-words‘ — Comments of users with an average amount of annotated images and an average percentage of correctly annotated images are considered incompetent if they contain any word indicating a question.
5. COMP: ‘ln-anno-img < 2‘ and ‘cor-anno > 0.9‘ — The data indicate that their are some expert users that have no or only very few annotated images. This rule correctly select those.
6. COMP: ‘ln-anno-img > 7‘ and ‘cor-anno > 0.9‘ — Comments of experienced users with a high percentage of correctly annotated users are considered competent
7. COMP: ‘ln-anno-img > 9‘ and ‘cor-anno > 0.85‘ and contains no ‘q-words‘ — Comments of very active users with an average to high number of correctly annotated images are considered as competent if they do not contain any word indicating a question.

Applying those rules to the Snapshot Serengeti user comments datasets results in the following statistics:

```
> category: null (count: 67396)
   - rule: Rule 0: 'no matching rule' (count: 39806)
   - rule: Rule 1: '< 3 tokens' (count: 27590)
> category: COMP (count: 14791)
   - rule: Rule 7: 'active users comment without query word'
      (count: 11965)
   - rule: Rule 6: 'high percentage of correctly annotated' (count:
      2669)
   - rule: Rule 5: 'expert user' (count: 157)
 > category: INCOMP (count: 11212)
   - rule: Rule 3: 'low percentage of correctly annotated' (count:
      6288)
   - rule: Rule 4: 'average users comment with query word' (count:
      2471)
   - rule: Rule 2: 'inexperienced user' (count: 2453)
```

This shows that by applying the rules one can create a training set containing more than 10k training examples for the two categories. This training set was used to train an OpenNLP Doccat model as required by the OpenNLP Classification Extractor. The competence classification also defines a simple thesaurus with two concepts representing the two categories so that the created fam:TopicAnnotation can use concepts URIs.

```
@prefix xsd: <http://www.w3.org/2001/XMLSchema#> .
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix dc:  <http://purl.org/dc/elements/1.1/> .
@prefix fam: <http://vocab.fusepool.info/fam#> .
@prefix mmm: <http://www.mico-project.eu/ns/mmm/2.0/schema#> .
@prefix oa: <http://www.w3.org/ns/oa#> .
@prefix compclass: <http://www.mico-project.eu/ns/compclass/1.0/schema#> .
@prefix example:   <http://www.example.org/mico/competenceClassification#> .

test:topic-classification-1 a fam:TopicClassification ;
 fam:classification-scheme compclass:CompetenceClassificationScheme ;
 fam:extracted-from test:item1 ;
 oa:item test:topic-annotation-2 , test:topic-annotation-1 .

test:topic-annotation-2 a fam:TopicAnnotation ;
 fam:confidence "0.2823413046977258"^^xsd:double ;
 fam:extracted-from test:item1 ;
 fam:topic-label "Kompetent"@de , "Competent"@en ;
 fam:topic-reference compclass:Competent .

test:topic-annotation-1 a fam:TopicAnnotation;
 fam:confidence "0.7176586953022742"^^xsd:double ;
 fam:extracted-from test:item1 ;
 fam:topic-label "Inkompetent"@de , "Incompetent"@en ;
 fam:topic-reference compclass:Incompetent .
```

The above listing shows the classification result of a user comment. The annotations include an annotation representing the classification as a whole and two TopicAnnotations describing that the comment

was classified to about 70% as incompetent and 30% competent.

In our experiments, the OpenNLP maximum entropy classifier reached an accuracy of 0.70 and an F1-score of 0.81. Looking closer at the results, however, these numbers are in part due to the fact that the classifier has a bias towards high proficiency and most of the test data forum posts were from high proficiency users. Still, the results are encouraging and we will continue developing these techniques.

Table 4 Competence Classification Extractor (TE-213)

Name	MICO Competence Classification Extractor
Original license	Apache Software License 2.0
MICO integration license	Apache Software License 2.0
External dependencies	This Extractor uses the OpenNLP Classification Extractor and so indirectly depends on OpenNLP
Input data	Plain text originating from written text or from speech to text transcoding
Output data	RDF
RDF persistence	Web annotation based annotations as defined by the MICO metadata model v2.0 with a single TopicClassification and possible multiple TopicAnnotation as defined by the Fusepool Annotation Modell[7]
External Parameters	none
Internal Parameters	The extractor is configured with the OpenNLP Doccat model trained on the Snapshot Serengeti user comments and a thesaurus representing the trained categories.

2.3.2.1 Specific comments

Performance: See performance notes of the OpenNLP Classification Extractor

Output: As all MICO NLP extractors, the OpenNLP NER extractor will use the MICO metadata model v2 and create annotation bodies for Topic Classification and TopicAnnotation as defined by the Fusepool Annotation Model. The Competence Classification Extractor only classifies whole documents, so it is most useful for short texts like comments or replays on forums.

2.3.3 Text Language Detection Extractor – New

This extractor detects the language of a text, e.g., English, French, and so forth. It is based on the Langdetect library. The correct detection of the language of a text is a pre-requirement of most NLP

processing as different languages required different configurations. Hence, the NLP processing components typically need to select the correct configuration for the language of the to be processed text.

The extractor consumes text/plain content from an asset detects the language of this textual content and writes a fam:LanguageAnnotation for the detected language

The following listing shows the output of this extractor.

```
@prefix fam: <http://vocab.fusepool.info/fam#> .
@prefix dc:  <http://purl.org/dc/elements/1.1/> .
@prefix mmm: <http://www.mico-project.eu/ns/mmm/2.0/schema#> .
@prefix oa:  <http://www.w3.org/ns/oa#> .
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix services: <http://www.mico-project.eu/services/> .
@prefix test: <http://localhost/mem/> .
@prefix xsd:  <http://www.w3.org/2001/XMLSchema#> .

test:item1 a mmm:Item ;
        mmm:hasAsset test:asset1 ;
        mmm:hasPart test:langAnnoPart ;
        mmm:hasSyntacticalType "text/plain" ;
        oa:serializedAt "2016-05-20 14:08:20.081" .

test:asset1 a mmm:Asset ;
        dc:format "text/plain" ;
        mmm:hasLocation "urn:eu.mico-project:storage.location:item1/asset1"
            ;

test:langAnno <http://purl.org/dc/terms/language> "en" ;
        fam:confidence 9.999980266137359E-1 ;
        fam:extracted-from test:item1 ;
        a fam:LanguageAnnotation .

test:langAnnoPart mmm:hasBody test:langAnno ;
        mmm:hasTarget test:e8bf6d0b-7add-412b-a9f3-220569fb2e25 ;
        a mmm:Part ;
        oa:hasBody test:langAnno ;
        oa:serializedAt "2016-05-20 14:08:20.421" ;
        oa:serializedBy services:text-lang-detect .

test:e8bf6d0b-7add-412b-a9f3-220569fb2e25 a oa:SpecificResource ;
        oa:hasSource test:item1 .
```

2.3.3.1 Specific comments

Performance: Both memory and processing footprint of this extractor are minimal. It does not use the whole textual content but randomly selected subsections of the content to detect the language. Because of that, the size of the textual content does not have a major influence on the performance of this extractor.

Output: Like all MICO NLP extractors, also this one creates RDF annotations according to the MICO metadata model v2 and the Fusepool Annotation Model. This extractor creates

Table 5 The OpenNLP Text Language Detection Extractor implementation (TE-213)

Name	MICO Text Language Detection Extractor
Original license	Apache Software License 2.0
MICO integration license	Apache Software License 2.0
External dependencies	This Extractor is based upon the Langdetect library.
Input data	Plain text originating from written text or from speech to text transcoding
Output data	RDF
RDF persistence	Web annotation based annotations as defined by the MICO metadata model v2.0 with one or more fam:LanguageAnnotation instances representing the languages detected for the processed text. Language Annotations are written as defined by the Fusepool Annotation Model[8]
External Parameters	none
Internal Parameters	none

fam:LanguageAnnotations for detected languages.

2.3.4 OpenNLP Named Entity Recognition (TE-220) – New

The extractor for Named Entity Recognition is based on Apache OpenNLP and the IXA Pipes [RAR14] extensions. For multilingual aspects, please see Section 2.4.

This extractor mines named entities from text/plain content. For this purpose, it requires the OpenNLP Name Finder models to be configured for different languages. Debian packages providing such models for English, German, Spanish and Italian to extract Persons, Organizations and Locations are provided by MICO. Other can be added by users.

Table 6 The OpenNLP Named Entity Recognition Extractor implementation (TE-220)

Name	MICO OpenNLP Named Entity Recognition (NER) Extractor
Original license	Apache Software License 2.0
MICO integration license	Apache Software License 2.0
External dependencies	This extractor will be based upon the OpenNLP frameworks and the IXA Pipes extensions (`http://ixa2.si.ehu.es/ixa-pipes/`)
Input data	Plain text originating from written text or from speech to text transcoding; fam:LanguageAnnotation for the text/plain asset - required to select the correct NER model
Output data	RDF
RDF persistence	Web annotation based annotations as defined by the MICO metadata model v2.0 with Named Entity annotations as defined by the Fusepool Annotation Model[9]
External Parameters	NER model location
Internal Parameters	The extractor will search for NER models in any sub-folder of the configured one. By default this folder is set to /usr/share/mico-extractor-opennlp-ner/models. This folder is also used by additional Debian packages providing the IXA NERC models for English, German, Spanish and Italian. For users that want to provide custom OpenNLP NER models it is recommended to create similar Debian packages providing such models

2.3.4.1 Specific comments

Performance: OpenNLP holds NLP models in memory. Using this extractor will require sufficient memory for loading all configured models in memory. Loading all NER models provided by MICO will require about 5 GB of RAM. In use cases where not all languages need to be supported Debian packages of other languages can be uninstalled to reduce memory consumption. OpenNLP can process concurrent requests on different CPU cores. Therefore scaling depends on the number and speed of available CPU cores.

Output: Like all MICO NLP extractors, the OpenNLP NER extractor uses the MICO metadata model v2 and create annotation bodies in accordance with the Fusepool Annotation Model.

Named Entity Recognition (NER) allows to detect mentions of trained entity such as Persons, Organization and Locations in texts. While OpenNLP comes with support for NER the quality of the default models is not sufficient for use in most application. The models distributed by IXA Pipes NERC provide much better quality. IXA Pipe NERC provides NER models for Basque, English, Spanish, Dutch, German and Italian. Named Entity Results will be represented by `fam:EntityAnnotation` annotations.

2.3.5 Redlink Text Analysis Extractor (TE-213, TE-220) – Update

Extractor that uses the Redlink Analysis Service part of the Redlink Semantic Platform to extract Named Entities; link to Entities defined in custom vocabularies or Wikipedia; sentiment analysis; classify texts along classification schemes and or extract keywords.

The Extractor itself is open source. The usage of the Redlink service requires an account for Redlink.

2.3.5.1 Specific comments

Performance: This extractor does only put minor load on MICO as the processing is done by the Redlink Analysis Service. The extractor itself sends the input content (text/plain) to the service and processes the received analysis results. Every request to the extractor will generate a request to the Redlink service. Those requests will count to the limit set for the account.

Output: The extractor outputs RDF according to the MICO metadata model (Open Annotation) and uses Annotations Bodies as defined by the Fusepool Annotation Model to annotate features extracted form the analyzed textual content.
The MICO metadata model v2 is supported starting with version 3.+ of the extractor. Older versions do use the MICO metadata model v1.

The Extractor supports the following types of annotations:

- Content Language (`fam:LanguageAnnotation`): annotates the language of the processed text.
- Named Entities (`fam:EntityAnnotation`): annotates named entities (person, organization, location and others) found in the text. The annotation also provides a selector with the exact position of the entity.
- Linked Entity (`fam:LinkedEntity`): annotates the mention of an Entity as define by a controlled vocabulary in the processed text. This is similar to a Named Entity, but also provide the reference (URI) for the detected entity.

Table 7 Text Analysis Extractor based on the Redlink Analysis Service (TE-213 and TE-220)

Name	mico-extractor-named-entity-recognizer
Original license	Apache Software License 2.0
MICO integration license	Apache Software License 2.0
External dependencies	Redlink Analysis Service(`https://my.redlink.io/`)
Input data	Text - while the Redlink platform supports plain as well as several rich text formats the extractor is currently limited to the media type 'text/plain'
Output data	The extractor outputs RDF only. No binary content part is added to the processed content item
RDF persistence	In version 3.+ the extractor does use the MICO metadata model v2. For Annotations it uses the annotation bodies for Named Entities, Linked Entities, Topic Classification, Sentiment Annotations and Keywords as defined by the Fusepool Annotation Model (`https://github.com/fusepoolP3/overall-architecture/blob/master/wp3/fp-anno-model/fp-anno-model.md`)
External Parameters	None
Internal Parameters	The Extractor need the 'Redlink Analysis Name' (-a) the 'Redlink Key' (-k) to be configured as parameters for the daemon. In addition the extractor allows to specify a 'Queue Name' (-q) parameter. This has to be used if multiple instance of this extractor configured for different 'Redlink Analysis' configurations can be addressed by the MICO broker.
Additional requirements	This extractor requires an account for the Redlink Analysis Service

- Topic Classification (`fam:TopicClassification` and `fam:TopicAnnotation`): annotation that classifies the processed text along topics defined by some classification scheme. Each `fam:TopicClassification` consists of one or more `fam:TopicAnnotation`s. The confidence of a topic annotation defines how well the processed text fits to the topic. If topics are defined by a controlled vocabulary the annotations will also provide a reference (URI) to the topic.
- Sentiment Annotation (`fam:SentimentAnnotation`): annotation that defines the sentiment of the processed text in the range of [-1..+1] where -1 sands for negative and +1 for a positive sentiment.
- Keyword Annotation (`fam:KeywordAnnotation`): keywords are words and phrases with a special improtance for the processed text. Keyword annotations provide a metric [0..1] that defines the importance as well as the count how often the keyword is mentioned in the processed text.

Note that the set of annotations extracted form the processed text will depend on the configuration of the configuration of the Redlink analyser application.

2.4 Multilingual support

Throughout the project, we have strived for broad multilingual support, with an emphasis on (i) the languages used in the MICO Showcases, and (ii) the official European languages. In the case of automatic speech recognition, we based our first pipeline on the Kaldi library (see Section 2.2.1 for a description of the central extractor). Kaldi represents the state of the art in speech recognition, is released under a business friendly license, and can be trained for any language for which there is a sufficiently large corpus of written and spoken text. On the downside, the availability of open-access language models for Kaldi is growing slower than expected. It is costly to train new models, so the companies that do are are not willing to share their results for free.

The MICO showcases require language support for English, Italian, and Arabic. Open-source models for English were easy to obtain, but for Italian we could only find proprietary ones, and non at all for Arabic. There are tools to convert language models from the older CMU Sphinx format to one suitable for Kaldi, but we were not satisfied with the results. We also tried to train our own language models for Arabic and Italian but were hampered by the lack of open training corpora. We did however manage to train a reasonable language model for the related language Amharic, which is the second-most spoken Semitic language in the world after Arabic. Both this model, and the training data and intermediate results for Italian and Arabic have been made available online for others to share.

To provide high-quality support for Italian and Arabic, we made an evaluation of the commercial alternatives. After some consideration, we decided to integrate Microsoft's speech recognition service Bing. Bing supports both Italian and Modern Standard Arabic, and in addition most of the official European languages. It allows for up to 20 free requests per minutes and has a state-less mode of interaction (which greatly simplifies integration). Most importantly, the quality of the transcripts were significantly higher that those produced by our open-sources language models. For more on this extractor, see Section 2.2.2.

Language support is also an issue for the textual language extractors, but for these tasks, there is a greater availability of multilingual tools. In our experience, it is also easier to train custom textual models than audio models. In MICO we have integrated functionality from the open-source libraries OpenNLP and Standford NLP, and the closed-source RedLink library.

2.5 Final extractor list

Table 8 Overview of all MICO extractors.

Name	Ver.	Lang.	Deb.	RDF	TEs	Purpose
Animal Detection HOG [2]	2.1.1	C++	yes	yes[1]	TE-202	annotation
Animal Detection DPM [2]	1.0.2	C++	yes	yes[1]	TE-202	annotation
Animal Detection Yolo (CNN) [2]	1.0.2	C++	yes	yes[1]	TE-202	annotation
Audio Demux [3]	2.2.1	C++	yes	no	TE-214	processing step
Face Detection [3]	2.1.0	C++	yes	yes[1]	TE-204	annotation
Diarization [3]	1.3.0	Java	yes	no	TE-214	processing step
Kaldi2rdf [2]	3.1.0	Java	yes	yes	TE-214	annotation help.
Kaldi2txt [2]	2.2.0	Java	yes	no	TE-214	processing step
Redlink Text Analysis (NER) [2,4]	3.1.0	Java	yes	yes	TE-213/220	annotation
ObjectDetection2RDF [2]	2.2.0	Java	yes	yes	TE-202	annotation help.
Speech-to-Text Kaldi [2]	2.2.0	C++	yes	yes[1]	TE-214	annotation
Speech-to-Text Bing (alpha) [2,4]	1.1.0	C++	yes	yes[1]	TE-214	annotation
Temporal Video Segmentation [5]	2.2.0	C++	yes	yes	TE-206	annotation
Media Quality [5]	discont.	C++	no	no	TE-205	annotation
Audio Editing Detection [5]	2.0.0	C++	yes	no	TE-224	annotation
Media Info [2]	2.0.0	C++	yes	yes[1]	TE-227	annotation
MediaTags2rdf [2]	0.9.0	Java	yes	yes	TE-227	annotation help.
Speech-Music-Discr. [5]	discont.	C++	no	no	TE-207	annotation
OpenNLP NER [2]	1.2.0	Java	yes	yes	TE-220	annotation
OpenNLP text class. Competence[2]	1.2.0	Java	yes	yes	TE-213	annotation
OpenNLP text class. Sentiment[2]	1.2.0	Java	yes	yes	TE-213	annotation
Text Language detection [2]	1.2.0	Java	yes	yes	additional	annotation
Stanford NLP	cancel.	Java	-	-	TE-213/220	annotation
Nudity Detection	cancel.	C++	-	-	TE-226	annotation
Generic Feature Extraction	cancel.	C++	-	-	TE-211/207	processing step
Video Segment Matching	cancel.	C++	-	-	TE-211	annotation

[1] via annotation helper
[2] Apache 2.0 license
[3] GPL license
[4] Apache 2.0 or GPL license but payed service used
[5] proprietary licences

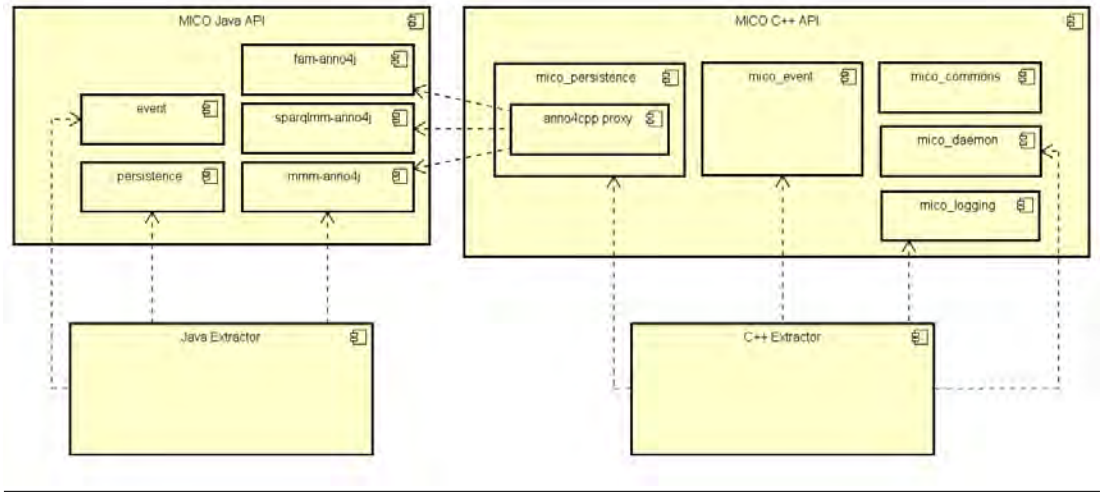
2.6 Extractor implementation guidelines – Broker v3

2.6.1 Extractor API Overview

MICO supports the implementation of media extractors in Java and C++. For both languages we provide APIs which we kept as consistent as possible among the two languages. Figure 1 shows a top level overview. The MICO APIs provide the following functionality:

- Runtime specific functionality (e.g. daemon tools for C++ extractors)
- Methods of communication with the MICO broker (registration, triggering analysis, error reporting, progress, finalization)

Figure 1 MICO API components overview



- Methods for annotation of the MICO meta data model (see also Section 4)
- Methods for extractor specific logging

The Java API is provided as artifact hosted on the MICO Nexus server. Due to the high amount of dependencies to other libraries and the difficulties in providing all these dependencies in a controllable way on a MICO Linux server we currently integrate the MICO Java API directly into the extractor artifact (creating fat jars). The C++ API is deployed as a system package (Debian package format) and re-used by any extractor.

2.6.2 Environment and Build

Although not a must, we strongly recommend to use CMake for C++ based and Maven for Java-based extractor services. Please make sure that dependencies to external libraries are properly documented (especially in the C++ case) and can be resolved by the build configuration.

2.6.3 OS Requirements

All extractors support the Linux (x64) operating system. Our current reference system is Debian 8.0 (Jessie). However, we have also successfully built the extractors on a Fedora 20 system (some dependencies must be installed manually though.). The Java extractors are basically compilable on any system with JDK 8, but since they are run as daemons, only Linux is supported during run time.

2.6.4 C++ Extractors

C++ extractors must follow these coding rules in order to be run by the MICO platform:

- Must be compiled as executables
- Must have mico-extractor-[name] as executable name

- Must provide **3 CLI positional arguments** in the following order and **1 CLI switch**:

  ```
  extractor_executable_name  [hostname] [username] [password] -k
  ```

  where

  [hostname] - is the server name of the mico platform (broker, marmotta, RabbitMQ)

  [username] - is the user name for the MICO platform

  [password] - is the user password for the MICO platform

  -k - is used to kill the daemon service in a controlled manner

  You may add additional CLI switches and options for the specific extractor configuration. Please also refer to C++ extractor examples in the repository.

- Must run as Linux daemons when executed. This can simply be achieved by using the mico-daemon provided by the MICO platform API. To start the daemon use `mico::daemon::start`. To stop it with the -k option use `mico::daemon::stop`. A minimal main() function could look like:

```
int main(int argc, char **argv)
{
  //...
  //set doKill according the command line parameter -k
  //...
  if(doKill) {
    return mico::daemon::stop("MyMicoExtractorDaemon");
  } else {
    return mico::daemon::start("MyMicoExtractorDaemon", server_name,
    mico_user, mico_pass, {new MyMicoExtractorService()});
  }
}
```

2.6.5 C++ Development Build

The extractor repository provides a module for finding the MICO platform C++ API for conveniences. Use

```
find_package(MICOPlatformAPI REQUIRED)
```

in your CMake-script to find it. If the MICO platform C++ API has not been installed into the Linux system you can give hints using `CMAKE_PREFIX_PATH` or `MICOPlatformAPI_HOME` as arguments to the cmake script. A typical call could then look like:

```
cmake -DMICOPlatformAPI_HOME=[path to MICO platform API]
    -DCMAKE_PREFIX_PATH=[path to local installs such as protobuf]
    [path to extractor source]
```

All dependencies introduced by the MICO platform C++ API should be resolved by the find script for the platform and using

```
target_link_libraries(${TARGET_NAME}
    #...
    ${MICOPlatformAPI_LIBRARIES}
)
```

in your extractors `CMakeLists.txt`.

Please check example extractors build configuration for more ideas on how configure the build.

Once you've successfully run CMake you may run

```
make -j
```

in your build directory to build your extractor.

2.6.6 Implementing your own C++ Extractor Service

The best way to learn how an MICO extractor works, is to look into the OCR example extractors located in the `api/c++/samples` directory of the MICO platform git repository. Here we give some explanations of what an extractor service needs to do in order to work within the MICO system.

In order to enable a new extractor service to be called by the MICO broker and to communicate its status it must derive from the `AnalysisService` class and implement the abstract `call` methods.

```
#include <AnalysisService.hpp>
#include <PersistenceService.hpp>

using namespace mico::persistence;
using namespace mico::persistence:model;
using namespace mico::event;

class MyFancyAnalysisService : public AnalysisService  {
  void call(AnalysisResponse& resp, std::shared_ptr< Item > item,
    std::vector< std::shared_ptr<Resource> > resources,
    std::map<std::string, std::string>& params);
};
```

The broker v3 (see Section 3.3) uses the extractors registration information and matches it to the running instance of an extractor in order to execute extractors routes. For this, the extractor must provide its **ID**, its current **runtime mode** and its **version**. In C++ this is done by calling the `AnalysisService` constructor in the extractors constructor.

```
AnalysisService(const std::string serviceID, const std::string requires,
    const std::string provides, const std::string queue)
```

For sake of easiness the extractor API provides method in the header file

```
#include <MICOExtractorID.h>
```

that automatically fill that information, provided that the related *precompiler flags* are available. These flags must be set in the CMakeLists.txt file as follows:

```
# returned by mico::extractors::getPrecompExtractorId()
add_definitions(-DMICO_EXTRACTOR_ID=${TARGET_NAME})

# returned by mico::extractors::getPrecompExtractorVersion()
add_definitions(-DMICO_EXTRACTOR_VERSION=${TARGET_VERSION})

# returned by mico::extractors::getPrecompExtractorModeId()
# omit when more than one mode is present
add_definitions(-DMICO_EXTRACTOR_MODE_ID=MyOnlyFancyMode)
```

A call to the base class extractor could thus look like:

```
MyFancyAnalysisService::MyFancyAnalysisService(...) :
  AnalysisService(mico::extractors::getPrecompExtractorId(),
    createModeId(inputMimeType),
    mico::extractors::getPrecompExtractorVersion(),
                    inputMimeType, "application/x-mico-rdf")
```

The last argument is the mime type produced by the extractor (assuming the extractor is producing RDF annotations, otherwise corresponding to the output format of the produced output) and kept for compatibility reasons to broker v2. Once the broker decides to send a Resource to process to the extractor, the `call` method is called which provide the following information:

- A response object that the extractor can use to communicate its status
- The original item ingested into the system
- The resources (Item and/or Parts) expected to be processed by the extractor (only one Resource in broker v2)
- a forth parameter to be used in broker v3

The first thing an extractor needs to do is to grab its input data e.g.:

```
std::shared_ptr<Asset> asset = resources[0]->getAsset();
std::istream* in = asset->getInputStream();
```

During processing the data broker v2 introduced two new ways to communicate the current extrator status:

- sending progress information. This is especially important if it takes longer to analyse an asset (e.g. a video). Example:

  ```
  resp.sendProgress(1.0);
  ```

- sending error information. This is is a huge benefit since now the broker is able to detect extractor internal errors. Example:

  ```
  resp.sendErrorMessage(item,mico::event::DECODING_ERROR,"Decoder init
     failed", "The decoder initialization failed for codec " +
     codec_type );
  ```

The following error types are currently defined by the MICO event protocol:

```
UNSUPPORTED_CONTENT_TYPE
UNSUPPORTED_CONTENT_VARIANT
DECODING_ERROR
MISSING_ASSET
MISSING_ANNOTATION
INSUFFICIENT_RESOURCE
INSUFFICIENT_STORAGE
UNEXPECTED_ERROR
```

Once the data in the input stream has been processed it time to

- Create a new part, if the extractor has produced one
- Annotate the **syntacticType** and **semanticType** of the new part
- Do semantic annotations if the extractors has produces some (see Section 2.6.7)

Creating new parts is rather simple since the persistence service provide native methods for that and does the required annotations in the MICO data model[10] using anno4cpp internally.

```
//creating the new part and pass extractor ID as creator
std::shared_ptr<Part> newPart = item->createPart(URI(getServiceID()));
std::shared_ptr<Resource> newResource =
    std::dynamic_pointer_cast<Resource>(newPart);

//get the asset - since it has just been created this produces a new asset
std::shared_ptr<Asset> asset = newResource->getAsset();

//get the stream of the asset
std::ostream* out = asset->getOutputStream();

//set the mime format of the asset
asset->setFormat("text/plain");

//write your produces data to the stream
*out << resultText;

//delete the stream
delete out;

/*
 * If no Body is created by the extractor
 */

//Set syntacticType and semanticType of the resource

//For binary data the syntactic type derives from the format, and usually
//corresponds to 'mico:Text' or 'mico:Video' or 'mico:Audio'
newResource->setSyntacticalType("mico:Text");

//Human-readable description of the output produced
newResource->setSemanticType("Plain text produced by our fancy extractor");

/*
 * Otherwise follow the example provided in the section
 * "Semantic Annotation via Anno4CPP"
 */

// notify broker that we created a new content part
resp.sendNew(item, newResource->getURI());
```

[10]For details, please refer to the MICO data model specification: `http://mico-project.bitbucket.org/vocabs/mmm/2.0/documentation/`

```
// notify broker that we done with processsing
resp.sendFinish(item);
```

2.6.7 Semantic Annotation via Anno4CPP

Since version 2.0, the C++ MICO API uses the Java features of anno4j (see 4.1) which is also developed within the MICO project. This approach makes sure that the MICO data model used by MICO Java and C++ extractors, the MICO broker as well as MICO endpoints and applications is consistent. On the other hand it introduces some complexity compared to a pure native implementation. Basically, the `PersistenceService` provides native wrappers taking care for annotations which are repeatedly required by the data model (Item, Part, Asset) as shown in the previous section. However, extractor specific annotations need to be done using the anno4cpp framework directly. This section will give some practical hints on how to to it and how to avoid pitfalls. Section 4.2 gives a short overview about the underlying principles of anno4cpp.

In order to use the anno4cpp proxies one must set the scope of the JVM to the current thread. An extractor developer must always do that since the call method is called asynchronously from different threads by the event manager:

```
#include <PersistenceService.hpp>
#include <anno4cpp.h>
//....
jnipp::Env::Scope scope(mico::persistence::PersistenceService::getJVM());
```

Then the following code illustrates the normal proceeding when using anno4j and the mmm-anno4j terms directly. Always make sure to check for Java exceptions, in particular after the final call. Instead of importing packages one can use C++ namespaces that are constistent with the Java packages. As long the object you create are used only in the local scope, you may use the `jnipp:LocalRef<T>` class for referencing them. If you plan to use them in different scopes, use `jnipp:GlobalRef<T>` and pass them through scopes via `jnipp:Ref<T>`. The following example of creating a face detection annotation illustrates the usage of anno4cpp.

```
using namespace jnipp::com::github::anno4j::model::impl::selector;
using namespace jnipp::eu::mico::platform::anno4j::model::impl::targetmmm;
using namespace jnipp::eu::mico::platform::anno4j::model::impl::bodymmm;
using namespace jnipp::eu::mico::platform::anno4j::model::namespaces;
using namespace jnipp::java::lang;
using namespace jnipp::org::openrdf::repository::object;

//create a FragmentSelector using anno4cpp
jnipp::LocalRef<FragmentSelector> jFragmentSelector =
    item->createObject(FragmentSelector::clazz());

//check for Java exception either by checking the return value of ...
if (resources[0]->getPersistenceService().checkJavaExceptionNoThrow()) {
  //do error handling
}
// ... or by throwing a C++ exception
resources[0]->getPersistenceService().checkJavaExceptionThrow();

//set properties of the fragment selector (region in this case)
```

```
jFragmentSelector->setSpatialFragment
  (Integer::construct(obj.region[0].posX),
    Integer::construct(obj.region[0].posY),
   Integer::construct(obj.region[1].posX-obj.region[0].posX),
   Integer::construct(obj.region[2].posY-obj.region[1].posY));

//create body and target
jnipp::LocalRef<BodyMMM> jBody =
    item->createObject(FaceDetectionBodyMMM::clazz());
jnipp::LocalRef<SpecificResourceMMM> jTarget =
    item->createObject(SpecificResourceMMM::clazz());

jTarget->setSelector(jFragmentSelector);

newPart->setBody(jBody);
newPart->addTarget(jTarget);

resources[0]->getPersistenceService().checkJavaExceptionThrow();

//Set syntacticType and semanticType of the new resource

//For RDF data the syntactic type corresponds to the Body Type, i.e.
newResource->setSyntacticalType(MMMTERMS::FACE_DETECTION_BODY->std_str());

//Human-readable description of the output produced
newResource->setSemanticType("Spatial region corresponding to the face
    detected inside the input asset");
```

2.6.8 Java Extractors

The implementation of Java extractors is similar to that of C++ extractors. It follows the same scheme and can use anno4j directly.

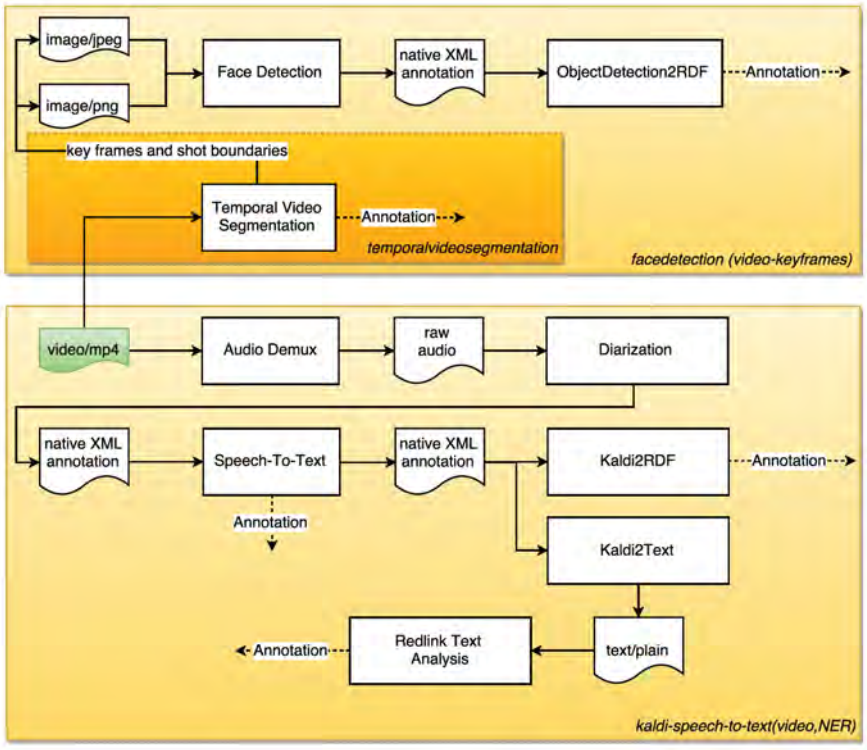
3 Enabling Technology Modules for Extractor Orchestration

3.1 Introduction

The *MICO broker* provides all components and tools necessary to orchestrate heterogeneous media extractors, thereby supporting complex analysis workflows (aka pipelines) which allow combination and reuse of otherwise disconnected results from standalone extractors, to achieve improved analysis performance. This sections provides information about the broker updates since MICO Technical Report Volume 3 (broker version 1), which were provided with version 2 and version 3 of the broker in the third year of the MICO project. It partially reuses material from a related publication at the LIME workshop.

To illustrate the challenges for extractor orchestration in MICO, an example workflow is depicted in Figure 2): It uses MP4 video containers as input, and consists of three partial workflows (see yellow/orange rectangles): (a) shot detection, and shot boundary and key frame extraction; (b) face detection, which operates on extracted boundary or key frames; (c) audio demux, speech2text and named entity recognition.

Figure 2 example MP4 video analysis workflow in MICO



The resulting annotations can be used for queries such as "Give me all shots in a video where a person says something about topic X", and of course, it could be extended with other extractors to further improve analysis performance, e.g., using speaker identification, face recognition, or extraction of metadata from the respective MP4 container, etc.

The key challenges for broker development in MICO arise from the question: What components,

protocols and data are required to support the creation and execution of such workflows? Based on early experiences and lessons learned in the project with the initial version 1 of the broker, the following key requirements for orchestration were identified:

1. *General requirements* including
    - backward-compatibility regarding the existing infrastructure (in order to reduce efforts for extractor adaptation, especially regarding RabbitMQ, but also the Event API wherever possible)
    - reuse of established existing standards / solutions where applicable; and support for several workflows per MICO system, which was not possible with v1
2. Requirements regarding *extractor (mode) properties and dependencies*:
    - support for extractor configuration, and support for different extractor "modes", i.e., different functionalities with different input, output or parameter sets, encapsulated within the same extractors
    - support for more than one extractor input and output; support for different extractor versions
    - distinction between different I/O types: MIME type, syntactic type (e.g., image region), semantic concept (e.g., human face)
3. Requirements regarding *workflow creation*:
    - avoiding loops and unintended processing
    - extractor dependency checking during planning and before execution
    - simplifying the workflow creation process (which was very complicated)
4. Requirements regarding *workflow execution*:
    - error handling, workflow status and progress tracking and logging
    - support for automatic process management
    - support for routing, aggregation, splitting within extraction workflows (EIP support)
    - support for *dynamic routing*, e.g, for context-aware processing, using results from language detection to determine different subroutes (with different extractors and extractor configurations) for textual analysis or speech2-to-text optimized for the detected language

The following will describe how these requirements were addressed with broker version 2 and the current version 3, which can be summarized as follows:

- v2 provided several major modifications and extensions to the MICO Event API related to error management, progress communication, provision of multiple extractor outputs, and registration. The goal of v2 was to provide an earlier API update for urgent issues and to give extractor developers an opportunity to already adapt to an extended broker data model to be applied for v3.
- v3, in contrast, was meant to focus on the addition of new broker components for registration, workflow creation and execution, to address most of the remaining aforementioned requirements.

The following will describe the current broker status and approaches in more detail, i.e. describe the broker model and components applied and describe how workflow creation and execution was implemented. Finally, we will also list some open issues that remain to be addressed in the future.

3.2 Broker Design and Implementation

3.2.1 Principles

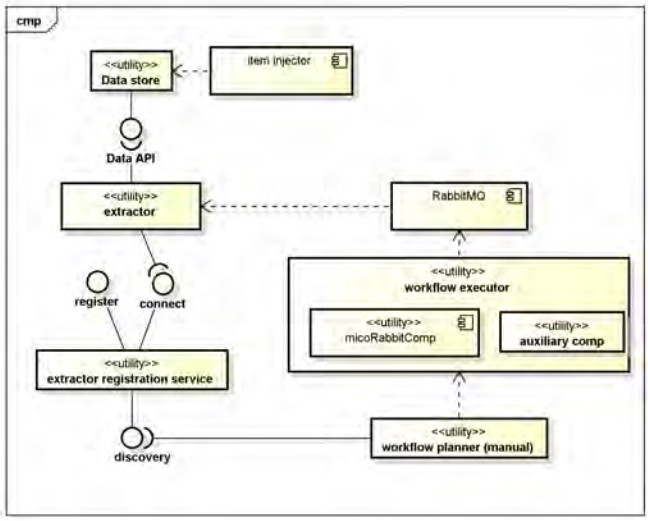
Broker v2 and v3 implementation follows a set of principles and assumptions, regarding **extractor registration and model**:

- Some parts of extractor information can be provided during packaging by the developer (extractor I/O and properties), while other parts can only be provided after packaging, by other developers or showcase administrators (semantic mapping, and feedback about pipeline performance): Registration information is provided at different times.
- Extractor input and output should be separated into several meta types (a) *MIME types* e.g., 'image/png', (b) *syntactic types* e.g., 'image region', and (c) *semantic tags* e.g., 'face region'. MIME types and syntactical types are pre-existing information that a extractor developer / packager can refer to using the MICO data model or external sources, while semantic tags are subjective, depending on the usage scenario, will be revised frequently, and are often provided by other developers or showcase administrators. Often, they cannot be provided at extractor packaging time, nor do they need to be, as they do not require component adaptation. As a consequence, different ways of communicating the various input and output types are needed.
- A dedicated service for extractor registration and discovery can address many of the mentioned requirements, providing functionalities to store and retrieve extractor information, supporting both a REST API for providing extractor registration information, and a front-end for respective user interaction, which is more suitable to complement information that is not or cannot be known to an extractor developer at packaging time.
- The MICO broker should reuse the existing MICO data model as far as possible, e.g., for syntactic types, and extend the MICO data model for that purpose; wherever applicable, extractors and extractor versions, types etc. should be uniquely identified via URIs; the broker also needs some specific information e.g. for workflow planning and execution, which is not relevant for the other MICO domains, and that should be persisted within the broker domain.

Regarding **workflow planning and execution**, we came to the following conclusions:

- Apache Camel is a good choice for workflow execution, supporting many EIP and all core requirements for the project. It should be complemented by MICO-specific components for retrieving data from Marmotta to put them into Camel messages, in order to support dynamic routing.
- The broker should not deal with managing scalability directly, but allow later scalability improvements by keeping information about extractor resource requirements, and allowing remote extractor process startup and shutdown.
- Manual pipeline creation is a difficult process, due to the many constraints and interdependencies, depending on the aforementioned types, but also content, goal, and context of an application. Considering this, we found that it would be extremely desirable to simplify the task of pipeline creation using a semi-automatic workflow creation approach that considers the various constraints. Moreover, it should store and use feedback from showcase admins on which extractors and pipelines worked well for which content set and use cases.
- Support of content sets and the mapping of content sets to workflows via *jobs* can be handy to simplify the execution and monitoring of workflows.

Figure 3 MICO broker components



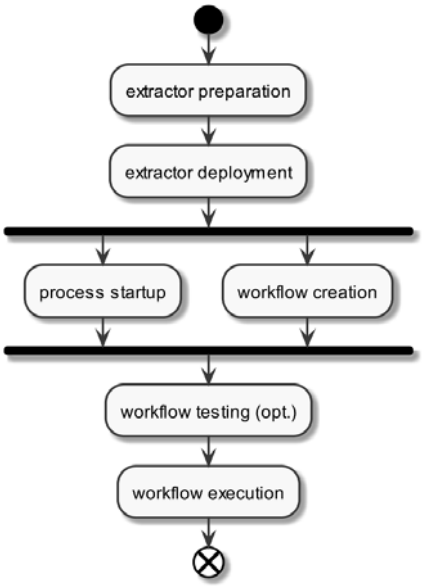
3.2.2 Broker Components

The broker includes the components depicted in Figure 3: The **registration service** provides a REST API to register **extractors** (which produce or convert annotations and store them as RDF), thereby collecting all relevant information about extractors. It also provides global system configuration parameters (e.g., the storage URI) to the extractors, and retrieval and discovery functionalities for extractor information that are used by the **workflow planner**. The workflow planner is responsible for the semi-automatic creation and storage of workflows, i.e., the composition of a complex processing chain of registered extractors that aims at a specific user need or use case. Once workflows have been defined, they can be assigned to content sets. Finally, the **item injector** is responsible for injecting items into the system, thereby storing the input data, and triggering the execution of the respective workflows (alternatively, the execution can also be triggered first on every extractor able to handle the input item, and then recursively on every other connected extractor, as in v1 and v2). Workflow execution is then handled by the **workflow executor**, which uses Camel, and a MICO-specific **auxiliary component** to retrieve and provide data from the data store to be used for dynamic routing within workflows. Finally, all aforementioned Linked Data and binary data is stored using the **data store**.

3.2.3 Extractor Lifecyle

From an extractor perspective, the high-level lifecycle can be summarized as depicted for the original broker design planning in Figure 4: **Extractor preparation** includes the preparation and packaging of the extractor implementation, including registration information that is used to automatically register the extractor component upon **extractor deployment** (and possibly test data and test case information for the extractor). As soon as extractor registration information is available, it can be used for **workflow creation**, which may include **extractor / workflow testing** if the required individual extractor test information is provided. For planning, or the latest for execution, the broker may have to perform an **extractor process startup** in certain cases, and **workflow execution** is then performed upon content

Figure 4 MICO extractor lifecylcle



injections or user request.

The current v3 of the broker implements all core functionalities of that process, i.e. everything except extractor and workflow testing support and remote process startup. In addition to the original planning, extra functionalities for content ingestion have been implemented, including (a) content set definition, (b) job definition (i.e. binding content sets to workflows) and (c) extractor (version and mode) availability checking and job execution, including user notification upon job completion, using a dedicated REST *workflow management* endpoint.

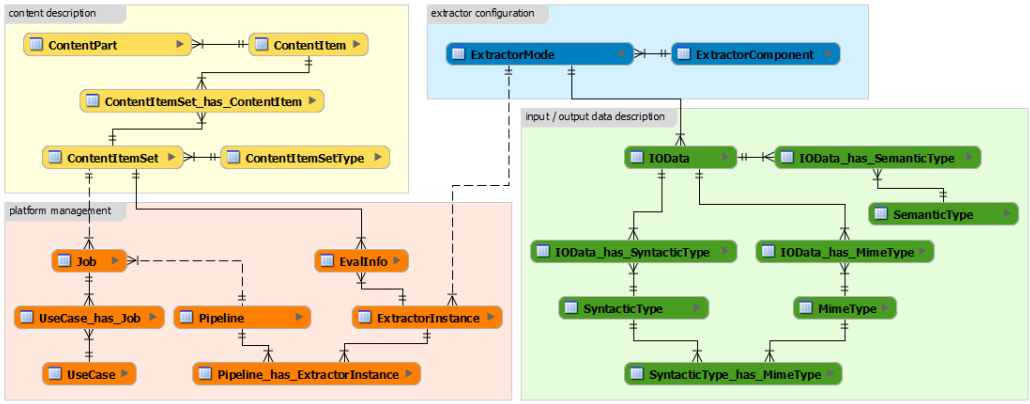
3.2.4 Broker Model

The data model of the MICO broker is meant to capture the information needed to to support extractor registration, workflow creation and execution, and collecting feedback from annotation jobs (i.e., processing workflows applied to a defined content set). It uses URIs as elementary data and extends the MICO Metadata Model (MMM)[11]. The broker data model is composed of four interconnected domains, represented by different colors in Figure 5, which are described in the following:

- The **content description** domain (yellow) includes the following main entities:
  - *ContentItem* captures information of items that have been stored within the system.
  - As described in [Aic+15, ch. 3.2], MICO items combine media resources and their respective analysis results in *ContentPart*.
  - *ContentItemSet* are used to group several *ContentItem*s into one *ContentItemSet*. Such a Set can be used, to run different pipelines on the same set, or to repeat the analysis with an updated extractor pipeline configuration.

[11] `http://mico-project.bitbucket.org/vocabs/mmm/2.0/documentation/`

Figure 5 MICO broker model overview



- The **extractor configuration** domain (blue) with two main entities:
  - *ExtractorComponent*, which captures general information about registered extractors, e.g., name and version
  - *ExtractorMode*, which contains information about a concrete functionality (there must be at least one functionality per extractor), which includes information provided by the developer at registration time, e.g., a human-readable description and configuration schema URI. For extractors which create annotations in a format different from RDF, it includes a output schema URI.
- The **input/output data description** domain (green) stores the core information necessary to validate, create and execute extractor pipelines and workflows:
  - *IOData* represents the core entity for the respective input or output to a given *ExtractorMode*.
  - *MimeType* is the first of three pillars for workflow planning. RDF data produced by extractors will be labeled as type *application/x-mico-rdf*.
  - *IOData_has_MimeType* connects I/O data to MimeType. It has an optional attribute *FormatConversionSchemaURI* which signals that an extractor is a *helper* with the purpose of converting binary data from one format to another one (e.g. PNG images to JPEG).
  - The *SyntacticType* of data required or provided by extractors is the second pillar for workflow planning. For MICO extractors which produce RDF annotations, the stored URI should correspond to an RDF type, preferably to one of the types defined by the MICO Metadata model([Aic+15, ch. 3.4]). For binary data, this URI corresponds to a Dublin Core format [Boa12].
  - *SemanticType* is the third pillar for route creation and captures high-level information about the semantic meaning associated with the I/O data. It can be used by showcase administrators to quickly pre-filter / discover new or existing extractors that may be useful for a their use case, even if the syntactical type is not (yet) compatible – this information can then be exploited to request an adaptation or conversion.

- The **platform management** domain (orange) combines several instances related to platform management:
  - *ExtractorInstance* is the elementary entity storing the URI of a specific instance of an extractor mode, i.e., a configured extraction functionality available to the platform. The information stored in the URI includes the parameter and I/O data selection and further information stored during the registration by the extractor itself.
  - *EvalInfo* holds information about the analysis performance of an *ExtractorInstance* on a specific *ContentItemSet*. This can be added by a showcase administrator to signal data sets for which an extractor is working better or worse than expected.
  - *Pipeline* captures the URI of the corresponding workflow configuration, i.e., the composition of *ExtractorInstance*s and respective parameter configuration.
  - *UseCase* is a high-level description of the goal that a user, e.g., showcase administrator, wants to achieve.
  - *Job* is a unique and easy-to-use entity that links a specific Pipeline to a specific Content Item Set. This can e.g. be used to verify the analysis status.
  - *UseCase_has_Job* is a relation that connects a *UseCase* to a specific *Job*, which can be used to provide feedback, e.g., to rate how well a specific *Pipeline* has performed on a specific *ContentItemSet*

As outlined in Section 3.2.1, a key broker assumption is that some extractor information is provided at packaging time by the developer (extractor properties, input and output) while other extractor information will typically be provided after packaging time, by other developers or showcase administrators. The registration service is one central point to register and query that extractor information, which provides the information needed for workflow planning (see Section 3.4) and execution (see Section 3.5). The registration service provides a REST API for providing extractor registration information.

3.3 Extractor registration service

The MICO registration service is the broker component responsible for storing and retrieving information about the extractors known to the MICO platform. This information, corresponding to the **extractor configuration** domain and to the **input/output data description** domain of the broker data model in Section 3.2.4, is of crucial importance for ensuring the compatibility of several extractor built independently from one another. Moreover, as described more in details in Section 3.4 and 3.5, is the core information used for both creating and executing the MICO workflows for complex analysis chains.

3.3.1 Example registration XML

The extractor registration information, as introduced in D2.2.2 [Aic+15], is firstly provided by the extractor developers in the form of an XML file. As defined in the XSD file previously presented in D2.2.2[12], the XML file contains a root element of type **extractorSpecification**, containing a unique *extractor id* and a respective *extractor name* for non-expert users, the *version* of the extractor, one or more *extractor modes*, and an indication about the extractor being a **singleton** – e.g., if requires a local, non-replicable database to run correctly:

[12] Available online at `https://bitbucket.org/mico-project/extractor-registration-beans/src/master/src/main/resources/micoRegistration.xsd`

```
<?xml version="1.0" encoding="UTF-8"?>
<extractorSpecification
    xsi:noNamespaceSchemaLocation="micoRegistration.xsd"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">

  <!--general info-->
  <name>Example Extractor</name>
  <version>1.0.0</version>
  <id>mico-example-extractor</id>

  <!-- mode ids must be unique! -->
  <mode>
    <id>first-mode</id>
    ...
  </mode>
  <mode>
    <id>second-mode</id>
    ...
  </mode>

  <!-- true if and only if only ONE instance at a time should be connected
    to the platform. In most of the cases, this is false -->
  <isSingleton>false</isSingleton>
</extractorSpecification>
```

Each **mode** of an extractor corresponds to a specific functionality – e.g., corresponding to a particular set of non-modifiable startup parameters – and thus also bound to a pre-defined set and amount of inputs. Every mode is specified by means of its **id** and human-readable **description**, one or more **inputs**, one or more **outputs**, zero or more **parameters**:

```
<extractorSpecification
    xsi:noNamespaceSchemaLocation="micoRegistration.xsd"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">

  ...

  <!-- mode ids must be unique! -->
  <mode>

    <!-- general info -->
    <id>Barack-Obama-Identification</id>
    <description>MICO extractors for identifying the face of Barack
    Obama</description>

    <!-- input definition-->
    <input>
       <!-- human readable information about the current input-->
       <semanticType>
         <name>Image with a detected face</name>
         <description>Image containing a face, it's going to be further
    analyzed to determine if it corresponds to Barack Obama</description>
```

```
18        </semanticType>
19        <!-- machine readable information -->
20        <dataType>
21          <!-- one entry for each allowed mimetype -->
22          <mimeType>image/png</mimeType>
23          <mimeType>image/jpeg</mimeType>
24          <!-- for binary data, this is one of mico:Audio , mico:Video ,
      mico:Image , mico:Text (mico:FirstPartOfTheMimeType)-->
25          <syntacticType>mico:Image</syntacticType>
26        </dataType>
27      </input>
28      <input>
29        <!-- human readable information about the current input-->
30        <semanticType>
31          <name>Face recognition body </name>
32          <description>Annotation associated with the input
      image</description>
33        </semanticType>
34        <!-- machine readable information -->
35        <dataType>
36          <!-- for rdf annotation, this is always -->
37          <mimeType>application/x-mico-rdf</mimeType>
38          <!-- rdf:type of the body assigned to the current input part-->
39          <syntacticType>mmmterms:FaceDetectionBody</syntacticType>
40        </dataType>
41      </input>
42
43      <!-- output definition -->
44      <output>
45        <!-- human readable information about the current input-->
46        <semanticType>
47          <name>Person</name>
48          <description>If present, signals the presence of Barack Obama in
      the input image</description>
49        </semanticType>
50        <!-- machine readable information -->
51        <dataType>
52          <!-- for rdf annotation, this is always -->
53          <mimeType>application/x-mico-rdf</mimeType>
54          <!-- rdf:type of the body assigned to the current input part-->
55          <syntacticType>http://example-onthology.com/v0.0.0#
      PersonIdentificationBody</syntacticType>
56        </dataType>
57        <!-- LDpath-based location of the output part-->
58        <location><![CDATA[mmm:hasPart [ mmm:hasBody / rdf:type is
      http://example-onthology.com/v0.0.0#PersonIdentificationBody
      ]]]></location>
59      </output>
60
61      <!-- parameter definition -->
62      <param>
```

```
      <name>Recognition threshold</name>
      <description>Minimum confidance value (in probability) to be scored
    before recognizing Obama. Defaults to 0.9</description>
      <primitiveType>
        <primitive>float</primitive>
      </primitiveType>
      <allowedRange>
        <allowedRange>
          <minIncl>0.75</minIncl>
          <maxIncl>0.95</maxIncl>
        </allowedRange>
      </allowedRange>
    </param>

  </mode>

  ...

</extractorSpecification>
```

3.3.2 REST API

The MICO registration service has several functionalities, that can be accessed by means of a dedicated REST API. The primary functions address the management of the aforementioned XML schemas:

- **POST** /add/extractor – Register a new extractor (via XML file upload - see schema micoRegistration.xsd)
- **POST** /update/extractor/{id:.+} – Replace an existing extractor specification - i.e. the whole document (via XML file upload)
- **GET** /get/extractor/{id}/xml – Provides information about a specific extractor (as XML)
- **GET** /get/extractor/{id}/json – Provides information about a specific extractor (as JSON)
- **DELETE** /delete/extractor/{id:.+} – Delete an extractor (and related deployments)

A further part of the REST API is completely dedicated to the retrieval of the available extractors, also depending on the required output, input, or on their human-readable name:

- **GET** /find/extractors – Returns all the available extractors
- **POST** /find/extractors/input – Find extractors requiring specific types
- **POST** /find/extractors/output – Find extractors providing specific types
- **GET** /find/extractors/name/{name:.+} – Find extractors having a specific name
- **GET** /find/extractors/mode/{keyword:.+} – Find extractors containing modes having a specific description

The third and last part of the REST API, is dedicated to the extractor deployments information – that is intended to be used, e.g., for the remote extractor startup required by the MICO broker:

- **POST** /add/deployment/{id:.+} – Add deployment information for a specific extractor
- **POST** /update/deployment/{id:.+} – Update deployment information for a specific extractor

GET /get/deployments/extractor/{id:.+} – Get deployment information for an extractor

DELETE /delete/deployment/{deploymentid}/extractor/{extractorid:.+} – Delete deployment information from a specific extractor

The complete description of the API, including data formats of the input required by POST operations, is currently available at `http://mico-project.bitbucket.org/api/rest/index.html?url=registration.json`.

3.4 Workflow Creation

One of the lessons learned of the initial MICO project phase was that manual creation of workflows is more complicated and difficult than expected, as it requires know-how not only regarding extractor inter-dependencies and constraints on multiple levels, but also regarding the content used, and regarding the context and goal of a specific analysis use case.

In order to address this problem, the idea of a semi-automatic workflow creation process emerged, by means of web-based graphical tool for progressive creation and management of MICO workflows.

3.4.1 Using the workflow creation tool

In a first step, the workflow creation tool retrieves information about all the extractors registered to the platform by accessing the registration service described in Section 3.3. Then, it finds possible combinations of matching extractors using two of the three *information pillars* outlined in Section 3.2, namely the **syntacticType** and **mimeType**.

The syntactic types are used to build a **first graph**, where all available MICO **extractor modes** (displayed as golden nodes) are connected between each other only if the respective input/output **syntactic types** (displayed as green nodes) are equal. The user can then select the extractors to be included in its workflow by clicking on the relative nodes, obtaining an initial configuration as in Figure 6. In this example, the creation process started from multimedia content with MP4 video and text, where the user could incrementally add suitable extractors proposed by the GUI using the aforementioned information pillars.

After selecting the desired set of extractor modes, the user can configure each of them separately, e.g. by selecting specific input and output **mime types** to be used, as well as by setting the desired values of the individual **run-time parameters**. The options are available by ad-hoc forms as in Figure 7.

After the selection of parameters and mime types, the user can verify that the connections between extractors can work as expected or not. For instance, Figure 8 shows a workflow that is validated, while Figure 9 shows the same but invalid workflow, due to a slightly different configuration of the AudioDemux: The latter does not provide the output of type “audio/wav”, which leads to an incompatibility that is signaled within the GUI.

Even from this small example, it should be clear that the broker information pillars do not represent a simple hierarchy: For instance, the indication of two extractors providing and consuming a matching *mimeType* and *syntacticType*, but lacking the same *semanticType* can be used to signal to the service which extractors *could* match and hence should be linked via a new *semanticType*, requiring human feedback to create this link. Vice versa, if it turns out that two extractors seem to provide similar output, as signalled by *syntacticType* and *semanticType*, but the *mimeType* does not fit – which could be the case for a workflow as in Figure 9 –, this can be exploited as a signal that a simple extension of the extractor to support a new *mimeType*, e.g., via format conversion, could do the trick to create interoperability.

Figure 6 A preliminary extractor workflow

Figure 7 Configuration of extractor run-time parameters

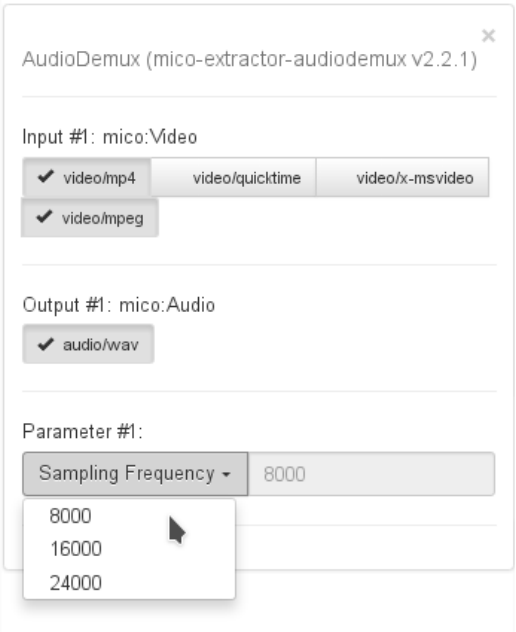


Figure 8 A correctly configured extractor workflow

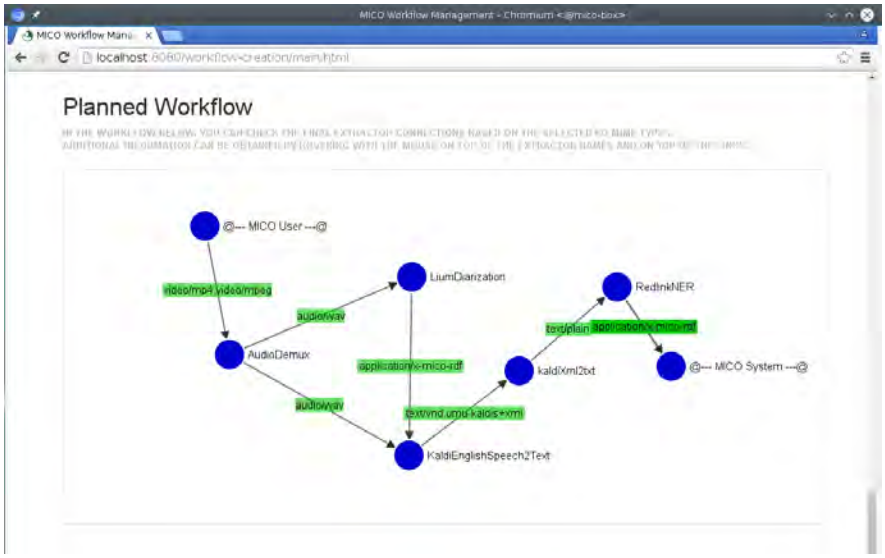
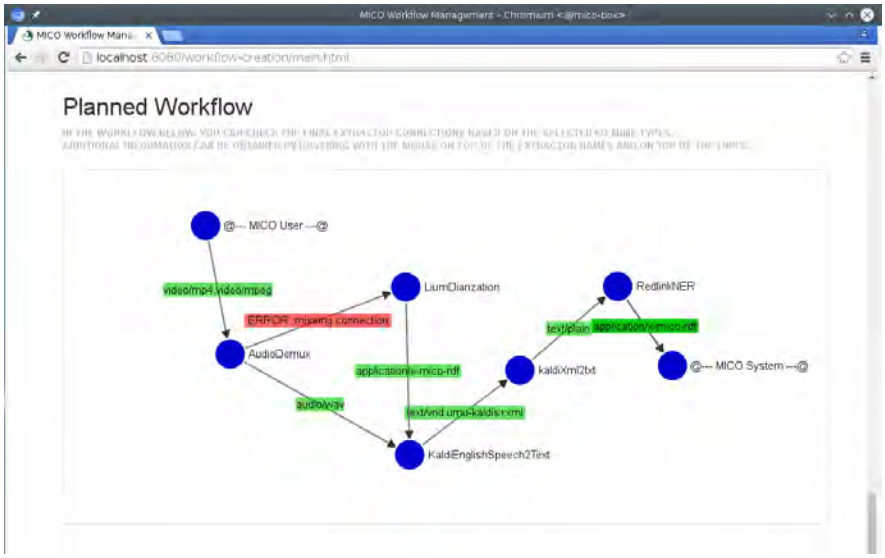


Figure 9 An extractor workflow with one missing connection



3.4.2 Camel Route definition

After completion, the user can store the resulting workflow as a Camel route to be used for workflow execution, e.g. as in Figure10.

Figure 10 Camel route example

These routes define the complete messaging of the an Item injected into the platform, and of every Part generated by the extractors, by means of routes, pipelines, splitters, multicasts and aggregators – i.e., by explicitly using robust Enterprise Integration Patterns provided by Apache Camel.

This choice, although producing complicated pipelines, is due to the peculiar requirements of the MICO extractors in terms of distributed processing, error messaging, and early stopping of items or parts that cannot be processed. Moreover, a filtering of the inputs both at syntactic and at mime type level is performed, thus ensuring the input/output data compatibility – which was not the case in the previous version of the broker.

An important remark, is that these extractor connections can be verified even *before starting any extractor implementation*. This is a crucial difference compared to the previous API, where the only way

of verifying data compatibility, was to *completely* implement the extractors involved in the pipelines.

The configuration chosen for the extractors is completely reported in the produced camel route, by specifying the related properties of the MICO RabbitMQ Camel component according to the configuration selected with the help of the forms:

```
<to uri="mico-comp:vbox1?
         extractorId=mico-extractor-audiodemux&amp;
         extractorVersion=2.2.1&amp;
         modeId=AudioDemux&amp;
         parameters={&quot;freq&quot;:&quot;8000&quot;}&amp;
         inputs={&quot;mico:Video&quot;:
                [&quot;video/mp4&quot;,&quot;video/mpeg&quot;]}"/>
```

3.4.3 REST API

The MICO broker includes a dedicated REST API for management and status retrieval of all the Camel routes deployed. A first set of of functions directly addresses the management of the XML camel routes:

POST /workflow/add – Add/replace all routes of a workflow to the broker
POST /workflow/del/{id} – Remove a workflow with a given id
GET /workflow/camel-route/{id} – Returns the XML representation of a specific workflow
GET /workflow/routes – Returns a list with all workflow IDs and relative descriptions

While a second set is dedicated to their *status retrieval*:

GET /workflow/routesInfo – Returns a list with all routes and their current status
GET /workflow/statusInfo/{id} – Return extended information about status of specific workflow

3.5 Workflow Execution

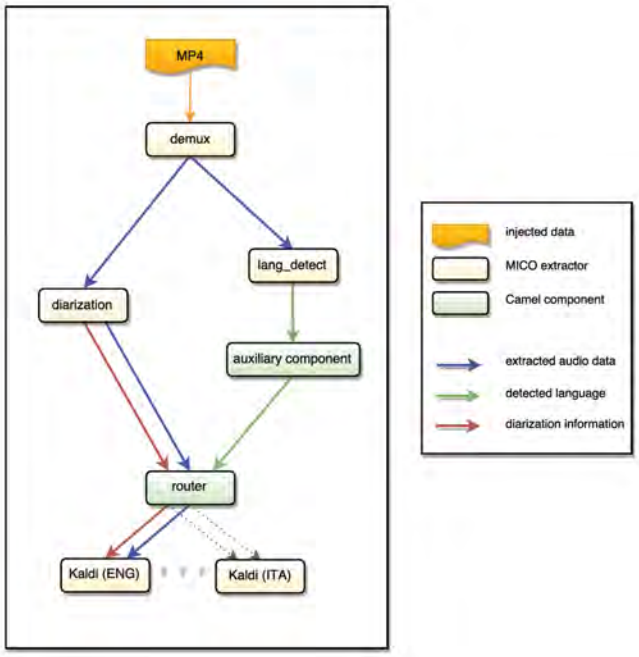
Once workflows have been created and stored as Camel routes, using the semi-automatic approach, they can be assigned to content items and sets, and execution can be triggered via the item injector or the user / showcase admin.

The actual workflow execution is performed using four main components, two of which were already mentioned in Section 3.2.2): The *workflow executor* as master component, which uses the other components and is based on Apache Camel, and the *auxiliary component*, a MICO-specific extension to Camel that allows Linked Data retrieval to support dynamic routing. In addition, the *RabbitMQ* message broker serves as communication layer to loosely couple extractors and the MICO platform, and a MICO-specific Camel endpoint component that connects Camel with the MICO platform, and triggers extractors via RabbitMQ.

An example for dynamic routing is depicted for extracting spoken words from an mp4 video (figure 11): After audio demuxing (*demux*), the audio stream from the mp4 video is stored and provided to *diarization*[13] and language detection (*lang_detect*. Both analyze the audio content in parallel, and store their annotations in Marmotta. At this point, dynamic routing is applied to optimize performance: The *auxiliary component* loads the detected language from Marmotta and puts it into the Camel message

[13]The segmentation of audio content based on speakers and sentences.

Figure 11 MICO workflow execution with dynamic routing



header – it knows where to locate the detected language, as *lang_detect* described by the storage location with its registration data via LDPath [Sch+12]. Afterwards, the language information within the Camel message header is evaluated by a *router* component, which triggers the *Kaldi* extractor optimized for the detected language. Beyond this example, there are many use cases where such dynamic routing capabilities can be applied.

3.5.1 Extractor and workflow status information

As shown in Table 9 the broker offers REST methods to check the current state of extractors. The following list explains possible states.

CONNECTED: at least one running instance of this extractor is connected to the broker
DEPLOYED: the extractor is registered but no running instance is connected to the platform. The extractor can be started by the broker
NOT_DEPLOYED: the extractor is registered but no running instance is connected and the broker is not able to start one
UNREGISTERED: an extractor with this id and version is not registered (is unknown) to the system

Depending on the extractor states involved in a workflow each workflow also has a state, which allows to broker distinguish if it can process items with or not. The following list explains the possible workflow states.

ONLINE: all extractors involved in this route are registered and connected
RUNNABLE: all extractors involved in this route are registered, but at least one is not connected. The missing extractors can be started by the broker, to bring the route online.

Table 9 Broker inject API.
Please see `http://mico-project.bitbucket.org/api/rest/?url=broker.json` for a detailed API documentation and all parameters. On a default configured MICO platform, the endpoints are available as a sub resource of `http://mico-platform:8080/broker`.

Method	*URL*	*Description*
POST	/inject/create	Create a new item and return its URI in the 'uri' field of the JSON response
POST	/inject/add	Add a new part to an item using the request body as asset content. The URI of the new part in the 'uri' field of the JSON response
POST	/inject/submit	Submit an item for analysis by notifying the broker to process it with a given workflow.
GET	/inject/items	retrieve information about injected items and corresponding parts
GET	/status/services	retrieve a list of currently connected extractor services
GET	/status/download	retrieve the binary asset of an item or a part
GET	/status/info	retrieve general information about the service as plain text

UNAVAILABLE: all extractors involved in this route are registered, but at least one is not connected and deployed on platform. To make this route available the missing extractor needs to be started manually by a user.

BROKEN: at least one extractor is not registered anymore. The route should be updated or removed from platform.

3.5.2 RabbitMQ Endpoints

In broker version 3 we adapted some parts of the initial rudimentary workflow orchestration approach, which simply send all new parts to all extractor services that could handle it, to use the camel framework [Fou04] for workflow definition and execution. More information about drawbacks of the old orchestration and the benefits of the new approach are explained in Section 3.2.1. To use camel framework within the MICO platform some MICO related camel extensions like a RabbitMQ [PS04] endpoint and aggregation components where implemented. The main task of the RabbitMQ endpoint is communication with the extractor services through the RabbitMQ message broker.

3.5.3 REST API methods for item creation and injection

Besides a web-front-end for item creation and process execution the MICO broker includes a REST API to be used remotely by other services. Table 9 provides a brief overview of the functions provided by Broker REST API. Further information like extended parameter and response description are available at [MP15]

A demo service, implemented for platform evaluation purposes, that uses the REST API to create and process new items is available in public git repository[14] of mico platform.

[14] `https://bitbucket.org/mico-project/platform`

3.6 Content Set, Job and Workflow Management

During the third year of the project, we developed an additional webservice, namely the MICO management service, meant to support the broker and provide advanced management functionalities for the showcase administrators of the MICO platform.

As thoroughly explained in the previous section, the MICO broker is addressing the execution of a *single* item against a *single* workflow. By means of the data model in section 3.2.4, however, the potential of the broker can be disclosed far beyond that point, e.g. by introducing the concept of **ContentSet** – i.e., a set of item or a set of item bundles – and of **Job** – i.e., the execution of every item/bundle of a ContentSet against a specific set of workflows.

This components, which were developed separately and in parallel to the broker, are going to be provided in a first step as closed-source freeware. They are not required for the platform execution, but are only meant to enhance its usability – and were hence implemented as REST-based web applications.

3.6.1 Workflow Management: REST API

The workflow manager provides persistence and retrieval of workflows – in contrast to the volatile in-memory behavior of the broker:

POST /add/workflow – Persist a camel route, and associated UI parameters
GET /get/workflow/status/{id} – Get the status of an existing workflow
GET /get/workflow/camel-route/{id} – Get the xml camel route definition of an existing workflow
GET /get/workflow/ui-params/{id} – Get the UI parameters of an existing workflow
DELETE /del/workflow/{id} – Delete a camel route
GET /list/workflows – Get a list of the existing workflows ids belonging to the current user
POST /resubmit/workflow/{id} – Convenience method for re-submitting an existing workflow to the broker (e.g., if the broker restarted and lost all camel deployed routes)

Further information can be retrieved by accessing the online documentation at `http://mico-project.bitbucket.org/api/rest/?url=management.json`.

3.6.2 Content Manager: REST API

The content manager provides methods for managing items and item sets. In addition to creating item sets only form local resources, is it possible to generate them by providing the URL to the respective elements to be download.

A first part of the API is addressing the management of single items:

POST /add/contentitem/public/{public} – Register a single content item. If public is set to true, also other users can include the new item to their own content sets
GET /get/contentitem/{uid}/{cid} – Provides info about a content item
DELETE /delete/contentitem/{uid}/{cid} – Delete a content item belonging to a user
GET /ispartofset/item/{uid}/{cid} – Checks if an item is part of any content set
GET /get/items/user/{uid} – Provides a list of available items, i.e. owned by user or public, both in groups and outside

A second part of the API is instead addressing the management of online and offline content sets:

POST /create/contentset – Create a new 'offline'-content-set

POST /add/contentset – Register a new online-content-set (via XML file upload - see schema OnlineContentSet.xsd)

POST /add/contentset/json – Register a new online-content-set (via JSON file upload)

GET /get/status/contentset/{id:.+} – Provides the processing status of a content set

DELETE /delete/contentset/{csid} – Delete a content set

GET /get/contentsets/user/{uid} – Provides a list of content sets (ids) which can be accessed by the specified user

GET /get/contentset/{id}/xml – Provides information about a content set (as XML)

GET /get/contentset/{id}/json – Provides information about a content set (as JSON)

GET /get/access/contentset/{csid}/user/{uid:.+} – Provides the access status for a specific content set and user

GET /get/groups/contentset/{csid}/user/{uid:.+} – Provides a list of groups belonging to a set and a certain user

GET /get/items/contentset/{csid}/group/{gid}/user/{uid:.+} – Provides a list of items belonging to a group of a specific set

Valid offline content sets can be configured with a simple JSON obect formatted as follows:

```
{
  "ownerId": 23,
  "description": "Example offline dataset",
  "isPublicSet": true,
  "contentIds": [
      "0/fb4c5cb6-6b09-4c19-aa43-fad8f8b34e62",
      "23/fd5957f2-ad41-4fd0-b1e3-24db84b0afca"
  ]
}
```

An example XML definition of an online content set is instead the following:

```
<?xml version="1.0" encoding="UTF-8"?>
<ContentSet Online="true" id=""
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:noNamespaceSchemaLocation="ContentSet.xsd">

  <Properties CreateSubsetOnFailure="true"
    CreationDate="2001-12-31T12:00:00" Description="Example online set"
    IsPublic="true" NotificationType="my.email@server.com" Owner="0" />
  <DownloadInfo>Created accessing the website of the EU MICO
    project</DownloadInfo>

  <Group GroupID="">
    <Content ContentID=""
    OriginalLocation="http://www.mico-project.eu/wp-content/uploads/2016/03/
    bird.png" StorageID="" Owner="" />
    <Content ContentID=""
    OriginalLocation="http://www.mico-project.eu/wp-content/uploads/2016/03/
    facedetect.jpg" StorageID="" Owner="" />
  </Group>
```

```
11
12  </ContentSet>
```

Further information can be retrieved by accessing the online documentation at `http://mico-project.bitbucket.org/api/rest/?url=management.json`.

3.6.3 Job Manager

A third part of the REST API addresses the Job Management:

GET /get/jobs/user/{uid:.+} – Provides a list of job ids available for a specific user

GET /get/job/{jid} – Provides information about a certain job

POST /add/job – Register a new job (via XML file upload - see schema Job.xsd)

GET /hasresults/job/{jid} – Checks if a job has a job result

DELETE /delete/job/{jid} – Delete a job

GET /ispartofjob/contentset/{csid} – Checks if a set is part of any job

GET /get/elements/job/{jid} – Provides a set of job elements for a certain job

POST /add/results – Add a result for a job

DELETE /delete/results/{jid} – Delete a job result

GET /start/job/{jid} – Start the job execution

GET /get/status/job/{jid:.+} – Provides the processing status of a job

POST /update/status/job/{jid} – Updates the status of a job - to be called by the broker

An example XML job definition is the following:

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <Job jid="" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:noNamespaceSchemaLocation="Job.xsd">
3    <JobProperties CreationDate="2001-12-31T12:00:00" Owner="0"/>
4    <JobElement csid="ContentSetID" wfid="WorkflowID"/>
5  </Job>
```

Further information can be retrieved by accessing the online documentation at `http://mico-project.bitbucket.org/api/rest/?url=management.json`.

3.7 Summary and Outlook

This section has outlined the challenges and status of the MICO broker, which includes solutions for all of the core requirements mentioned in Section 3.1.

Within the remaining project lifetime, it is to be expected that several issues will emerge from further testing based on the showcases demands, which will require further improvements of the broker. This seems realistic considering not only the various showcase demands and the fact that additional showcase demos have been planned in the meantime, but also the fact that there were many recent modifications of the overall platform e.g. related to persistence, MICO model and event API.

Beyond the core functionalities, there are also many ideas for possible improvements which go (far) beyond the original work planning, including

- advanced features for automatic process startup and management for scalability and distributed resource allocation
- improved inclusion of human interaction during workflows (annotations, feedback, etc.)
- improvements regarding security and access control for content
- front-end for providing extractor information that cannot be provided at packaging time, including feedback on extractor performance (complements the REST registration service)
- additional approaches regarding (semi-automatic) workflow creation
- full support for extractor and integrated workflow testing
- advanced workflow monitoring

Apart from the technical challenges, a key question for the MICO project will be on how to ensure further support and development in these domains by project partners beyond the project lifetime, also considering that there will be diverse demands especially from different commercial showcases.

4 Enabling Technology Modules for Cross-media Publishing

The recombination of various extraction results on one given multimedia file is a way of finding hidden semantics. These can lead to a richer metadata background which allows the respective file to be used in an even broader spectrum. Oftentimes however, this is not the case, as most multimedia analysis components work in isolation and do not consider the input of other extractors. The MICO platform allows to deal with this shortcoming by providing the possibility of orchestrating sets of extractors in order to jointly analyse the same content, which then can lead to further metadata discovery. Rather than publishing and storing the intermediary and final results in different proprietary formats, every extractor in a MICO workflow publishes its results using the Resource Description Format RDF[15]. This allows semantic interlinking and comprehensive querying with SPARQL. As a specifically for this use case designed RDF ontology, the MICO Metadata Model MMM[16]. The model is based on the W3C Web Annotation Data Model WADM [17]; the second iteration of the model was introduced in report V4. It allows to interlink the analysis results and track provenance over all workflow steps.

To facilitate the use of the MMM in the MICO platform, a software approach for an Object-to-RDF (ORM) mapping was created. The open-source library Anno4j[18] [Ber+16], which was already introduced in report V3, allows the creation of MICO or generic W3C Web Annotations via plain Java objects. Extensible querying is enabled using the path-based query language LDPath[19], which is described in a later section.

This section is structured as follows. Section 4.1 will give an in-depth description for the ORM library Anno4j. It will cover persistence, partial behaviour implementation, querying, and additional features added to the library. Afterwards, Section 4.2 introduces Anno4CPP, a conversion of the Anno4j library to the C++ language, enabling C++ extractors to use the same functionality. The ideas are explained considering the applicability for the MICO use case. Section 4.3 illustrates an extension to the MICO Metadata Model, showing the RDF classes, properties, and relationships that have been added in order to describe MICO extractors in a more detailed fashion. The last section, Section 4.4, will then list ideas that arose from various lessons learned during the MICO project.

4.1 Anno4j - An Object to RDF Mapping

An initial hurdle for extractor developers is to familiarize themselves with the Semantic Web technologies and the MICO Metadata Model, which are necessary to integrate valid extractor results into the MICO platform. Anno4j helps to overcome these problems by supporting the aforementioned ORM. By utilising the mapping, extractor writers can both persist and query WADM or MMM conform annotation results, without handeling the RDF content itself. The main "communication" is done via Java POJOs. The library has been designed in a modular and extensible fashion, to support enhancements and use-case specific model alterations, while the plugin functionality of Anno4j allows the enrichment of querying by adding custom function evaluators. Anno4j follows natural Object-oriented idioms including inheritance, polymorphism, and composition to facilitate the development of RDF ontologies, like the MICO Metadata vocabulary. The following list summarises the core functionalities of Anno4j:

- **Persistence**: Simple Java objects provide the basis of persistence and can easily be created and persisted with a given Anno4j object (see Section 4.1.1). By persisting an Anno4j Java object,

[15] https://www.w3.org/TR/rdf11-concepts/
[16] http://mico-project.bitbucket.org/vocabs/mmm/2.0/documentation/
[17] https://www.w3.org/TR/annotation-model/
[18] https://github.com/anno4j/anno4j
[19] http://marmotta.apache.org/ldpath/

corresponding RDF triples are created. This "basic persistence" can be extended in order to support behaviours to the Java interfaces (see Section 4.1.2).

- **Querying**: A `QueryService` object created by an Anno4j instance can be augmented with different query criteria (formalised as LDPath arguments) to query and consume respective RDF data of the underlying SPARQL 1.1 endpoint (see Section 4.1.3). The response is turned into respective (single or multiple) Java POJOs for further convenient usage.

- **Transaction support**: Transactional behaviour can be used to create sets of operations, which can either be fully committed or not at all (see Section 4.1.4).

- **Context awareness**: RDF databases are often using different contexts to divide their data into subgraphs. This feature is also possible in Anno4j, turning RDF triples into quads (see Section 4.1.4).

- **Plugin Extensions**: By supporting a plugin interface, users can define own LDPath functions in combination with respective evaluation operators. These can be used as querying criteria in order to even enhance the querying functionality and fine tune it to their particular use-case (see Section 4.1.4).

- **Input and Output**: Anno4j is able to both read and write annotations from and to different standardised serialisations, such as JSON-LD, TURTLE, N3-Triples, RDF/XML, etc (see Section 4.1.4).

- **MMM, MMMTerms & WADM implementations**: Built-in and predefined implementations for classes of the MICO Metadata Model and the W3C Web Annotation Data Model enable easy access to respective ontologies and allow a quick start right away. MICO workflow specific body implementations are included in the MMMTerms vocabulary.

4.1.1 Anno4j Persistence

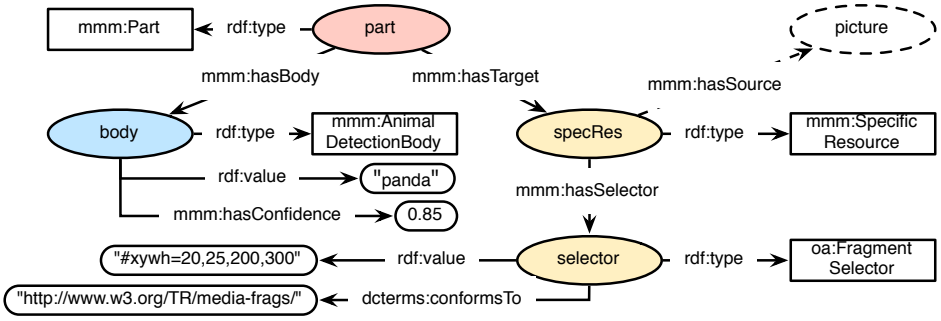
Anno4j supports an object-relational-mapping (ORM) between RDF objects and Java POJOs. This is mainly based on the Alibaba[20] library, formerly known as the Elmo codebase, which is responsible for the internal transformation between RDF and Java. Smaller tweaks have been made in order to adjust the library to the requirements of MICO.

This section will describe how to do one direction of the ORM: from Java POJOs to RDF. Via Anno4j, respective Java objects are turned into RDF automatically. Utilising this procedure, it is possible to create one's own RDF model - classes, properties, and relationships - and persist them. The other direction, from RDF to Java POJOs in order to query for already stored data, is shown in Section 4.1.3.

Throughout the rest of this Anno4j documentation, we will stick to the exemplary use case of an animal detection process. Therefore, we will produce annotations which indicate **what animal** has been detected on a given image and the degree of **confidence** of the extractor about that statement. Figure 12 shows an example of a complete MICO Part annotation with the content of an animal detection. In this example, a **panda** is found with a confidence of **85%**. With its `selector`, only a fragment of the supported `picture` is to be targeted by the Part annotation. It's value is **#xywh=20,25,200,300**, which corresponds to the fragment starting at top left pixel with the position **(20,25)** with a width of **200** pixels and a height of **300** pixels.

[20] `https://bitbucket.org/openrdf/alibaba/`

Figure 12 Exemplary RDF graph for an animal detection MICO Part annotation. This respective result indicates that a **panda** was detected with a confidence of **0.85** on the given `picture`.



Anno4j supports implementations for most of the classes needed by the Web Annotation Data Model WADM. On top of that, the MICO adaption adds all remaining classes, that were introduced by the MICO Metadata Model MMM[21] in order to produce the metadata of MICO workflows. Respective body implementations can be found at the MICO Metadata Model Terms vocabulary[22]. Every RDF node is implemented as an interface class in Anno4j. When creating an instance of it, the proxy pattern is applied and a proxy object of the respective interface is generated. An interface can be implemented in order to create a subclass, which is also reflected in the RDF graph.

When introducing a new kind of extractor or partial result for a given MICO workflow, the main thing to implement for the persistence layer is a body class. Referring to the exemplary animal use case, a body class with a **type**, a String field for the **detected animal**, as well as a double field for the given **confidence** is needed. Anno4j supports the body interface (`BodyMMM`) that needs to be implemented in order to be a body node. For every field, a getter/setter pair needs to be supported. Listing 1 shows the basic interface frame that is needed for the desired animal detection body.

Listing 1: Basic interface frame for an animal detection body

```
// Interface for the animal detection body
public interface AnimalDetectionBody extends BodyMMM {

                // Getter/setter pair for the confidence
    void setConfidence(Double confidence);

    Double getConfidence();

                // Getter/setter pair for the detected animal
    void setAnimal(String value);

    String getAnimal();
}
```

[21] http://mico-project.bitbucket.org/vocabs/mmm/2.0/documentation/
[22] http://mico-project.bitbucket.org/vocabs/mmmterms/2.0/documentation/

In Listing 1, Line 2 shows the interface definition needed for the `AnimalDetectionBody`. The definition must be an interface and must extend the `BodyMMM` interface (`BodyMMM` itself is an extension to the basic `Body` class in Anno4j. This is an extension done by the MICO adaption to the WADM because of typing purposes). Line 5 and 7 define the getter/setter pair for the confidence, while Line 10 and 12 do the same for the String field describing the detected animal. Note that for a desired field one can use basic datatypes (and thereof only their wrapper classes) for the type of the field. The (Java-) naming of the pair is can be chosen freely.

The next important step consists of the creation of the RDF relation. This is done by adding the Java annotation `@Iri` at two specific locations to the interface:

- Above the *interface definition*: This defines the RDF type of the corresponding RDF node.
- At every *getter/setter pair*: This defines the relationship or property attached at instances of the given RDF node.

Listing 2 is an alteration of the basic interface from Listing 1, adding the Java annotations `@Iri` at three different occasions.

Listing 2: Basic interface extended with Iri annotations

```
// Iri refers to
   http://www.mico-project.eu/ns/mmmterms/2.0/schema#AnimalDetectionBody
@Iri(MMMTERMS.ANIMAL_DETECTION_BODY)
public interface AnimalDetectionBody extends BodyMMM {

                // Iri refers to
                   http://www.mico-project.eu/ns/mmm/2.0/schema#hasConfidence
                @Iri(MMM.HAS_CONFIDENCE)
    void setConfidence(Double confidence);

                @Iri(MMM.HAS_CONFIDENCE)
    Double getConfidence();

                // Iri refers to
                   http://www.w3.org/1999/02/22-rdf-syntax-ns#value
                @Iri(RDF.VALUE)
    void setAnimal(String value);

                @Iri(RDF.VALUE)
    String getAnimal();
}
```

In Listing 2, Line 2 defines the type (RDF relationship `rdf:type`) for the defined interface and respective RDF node. It is set to `http://www.mico-project.eu/ns/mmmterms/2.0/schema#AnimalDetectionBody`. The properties are defined to be of the relationship type `http://www.mico-project.eu/ns/mmm/2.0/schema#hasConfidence` (in lines 6 and 9) and `http://www.w3.org/1999/02/22-rdf-syntax-ns#value` (in lines 13 and 16) respectively. Note here: Anno4j supports different RDF vocabularies via namespace classes that contain constants for the namespace itself, its prefix, and all associated classes, relationships, and properties. The

MICO addition features a basic vocabulary for the MMM, as well as a vocabulary MMMTERMS featuring MICO specific implementations. In this example, the three Iris can be exchanged with `MMMTERMS.ANIMAL_DETECTION_BODY`, `MMM.HAS_CONFIDENCE`, and `RDF.VALUE` respectively.

In order to create the example Part annotation seen in Figure 12 in an Anno4j workflow, the code shown in Listing 3 can be used.

Listing 3: Anno4j workflow to create a animal detection part annotation

```
// Create Anno4j instance
Anno4j anno4j = new Anno4j();

// Create single nodes
PartMMM part = anno4j.createObject(PartMMM.class);

AnimalDetectionBody body =
    anno4j.createObject(AnimalDetectionBody.class);
body.setAnimal("panda");
body.setConfidence(0.85);

FragmentSelector selector =
    anno4j.createObject(FragmentSelector.class);
selector.setConformsTo("http://www.w3.org/TR/media-frags/");
selector.setValue("#xywh=20,25,200,300");

SpecificResourceMMM specRes =
    anno4j.createObject(SpecificResourceMMM.class);
// Set the source to the associated picture
specRes.setSource(... pictureURI ...);

// Join nodes together
part.setBody(body);
part.addTarget(specRes);
specRes.setSelector(selector);
```

4.1.2 Anno4j Partial Behaviour

An expressive feature that can be introduced in the Java-supported implementation of MICO's metadata storage is the establishment of **behaviours**. This behaviour can be executed before respective RDF instances are created. An example for this is the check if a given value lies inside a predefined range.

To show the functionality and how to implement it, we utilise the interface that was created in Section 4.1.1 for the animal detection, shown in Listing 2. The getter method to specify the confidence (`setConfidence(Double confidence)`) is to be extended with the behaviour to only set values that are between `0.0` and `1.0`. Values above or below those boundaries are to be set to the respective boundary value. The implementation is shown in Listing 4.

Listing 4: Support class implementing the `AnimalDetectionBody` interface shown in listing 2

```
@Partial
```

```
public abstract class AnimalDetectionBodySupport extends
    ResourceObjectSupport implements AnimalDetectionBody {

    @Iri(MMM.HAS_CONFIDENCE)
    private Double confidence;

    @Override
    public void setConfidence(Double confidence) {
        if(confidence < 0.0) {
            this.confidence = 0.0;
        } else if(confidence > 1.0) {
            this.confidence = 1.0;
        } else {
            this.confidence = confidence;
        }
    }

    @Override
    public Double getConfidence() {
        return this.confidence;
    }
}
```

Important things to note in the implementation shown in Listing 4, and also when implementing support classes altogether, are the following:

- **Partial Java Annotation**: The `Partial` annotation in Line 1 is necessary for project build reasons and for the scanning of behaviours for other interfaces.
- **Class Declaration**: As method bodies are needed in order to implement the behaviour for the methods, an abstract class is needed at this point.
- **Implementation and Extension**: Support classes implement the interface of which the behaviour is to be implemented. Also, it needs an extension of the support class of the next lowest super class (in this case, the `ResourceObjectSupport`).
- **Connection to the RDF Graph**: In order to establish the association with the corresponding RDF properties and relationships, the respective field has to be defined as private field of the abstract class. In the case of the example, Line 5 defines a Double field for the confidence, and the appropriate Iri is defined in Line 4.
  IMPORTANT NOTE: The Iris have to be removed from the implemented interface class! (in this case from the `AnimalDetectionBody` interface)
- **Overriding Methods**: The methods in the support class need to overwrite those defined in the implemented interface. Also, they can now implement the desired behaviour.

4.1.3 Anno4j Querying

The query feature of Anno4j implements the other direction of the ORM mapping between Java objects and RDF. While the persistence described in Section 4.1.1 turns created Java POJOs into RDF triples,

an executed query at an Anno4j object retrieves the RDF information stored at the Anno4j's triplestore and returns a set of Java POJOs.

The central Java class for the querying is the `QueryService`, which can be created at a respective Anno4j instance. Calling the `.execute()`-method of a QueryService creates a SPARQL query with the query criteria (see below) that are added to the QueryService instance, and dispatches it to the associated triplestore. A class parameter can be passed to the method in order to define the (RDF graph node) starting point for the query. Classic examples in the MICO context would be `.execute(ItemMMM.class)` or `.execute(PartMMM.class)` to query for Items or Parts respectively. The basic behaviour is to query for annotations (of the type `oa:Annotation`). The return type of the method is a Java List of the specified class. Listing 5 shows an example of how to create and execute a QueryService.

Listing 5: Creation and execution of a QueryService

```
Anno4j anno4j = new Anno4j();

QueryService qs = anno4j.createQueryService();
List<ItemMMM> result = qs.execute(ItemMMM.class);
```

The **criteria** mentioned above shape the definition of a query. One can add different requirements and confinements in order to fine tune the query and thereby specify which part of the RDF graph is to be requested. To do this, we decided to implement the criteria using the path-based query language LDPath (see Section 5 for a detailed description of LDPath). LDPath is well suited to query information on an RDF graph. Additionally, rather than writing whole extensive SPARQL queries, it seemed more convenient to non-RDF experts to define single criteria by writing LDPath expressions. Every LDPath criteria defines a path in the RDF graph, which can be enhanced by different requirements, in order to query for various nodes that comply with the defined criteria, a set of LDPath criteria is transformed into respective SPARQL 1.1 queries. By transforming the LDPath expression to SPARQL, every SPARQL 1.1 compliant endpoint can be registered as RDF storage with an Anno4j instance.

As mentioned before, the `QueryService` is the central point of Anno4j's querying feature. Once created, a user can add different criteria to the QueryService to define the respective query. In order to further enhance its usability, a fluent interface is implemented for the QueryService, allowing the user to add the query criteria bit by bit instead of having to define them all at once, which we assume to be more convenient.

However, before one can start to define queries, it is necessary to register all respective namespaces with its corresponding abbreviation. This enables to use the abbreviations instead of having to support the full URI every time. Like before, Anno4j already supports the common RDF standards, they must not be set beforehand. The predefined namespaces are listed in table 10.

To add your personal namespace, the `.addPrefix(String abbreviation, String uri)` method of a given `QueryService` is used. The namespace is then registered and can be used at various locations of the supported LDPath criteria. Adding a prefix can also be integrated in the fluent interface of the QueryService. An example can be seen in Listing 6, adding the namespace "**http://www.example.com/schema#**" with the abbreviation "**ex**".

Listing 6: Registering of a custom prefix at a QueryService

```
QueryService qs = anno4j.createQueryService();
qs.addPrefix("ex", "http://www.example.com/schema#");
```

In order to fine-tune a query, the already mentioned query criteria formalised as LDPath expression have to be added to the `QueryService`. Listing 7 shows an example, adding various query criteria to a

Table 10 Predefined Namespaces for a QueryService

Abbreviation	Namespace
oa	<http://www.w3.org/ns/oa#>
cnt	<http://www.w3.org/2011/content#>
dc	<http://purl.org/dc/elements/1.1/>
dcterms	<http://purl.org/dc/terms/>
dctypes	<http://purl.org/dc/dcmitype>
foaf	<http://xmlns.com/foaf/0.1/>
prov	<http://www.w3.org/ns/prov/>
rdf	<http://www.w3.org/1999/02/22-rdf-syntax-ns#>
rdfs	<http://www.w3.org/2000/01/rdf-schema#>
skos	<http://www.w3.org/2004/02/skos/core#>

given `QueryService`. The query relates to the animal detection results shown in Figure 12. Formulated as text, we want our query to be defined by the following requirements:

1. Query for Parts of the database only,
2. Query for results of the animal detector, so Part annotations with a body node of the type `mmm:AnimalDetectionBody`,
3. Query for pandas only,
4. Query for rectangles exactly of a width of 200 pixels and a height of 300 pixels (this criteria is rather unusual, it is merely used for exemplary purposes, more reasonable spatial or temporal logic can be introduced with plugins, see section 4.1.4).

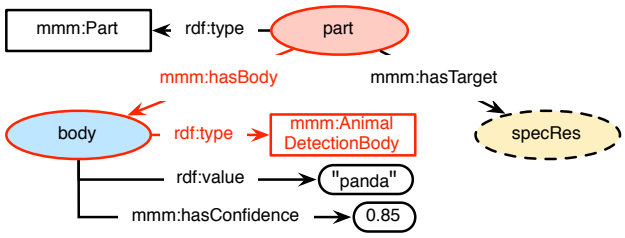
Listing 7: Addition of various query criteria to a `QueryService`

```
QueryService qs = anno4j.createQueryService();
qs.addPrefix("mmm", "http://www.mico-project.eu/ns/mmm/2.0/schema#")
        .addCriteria("mmm:hasBody[is-a mmm:AnimalDetectionBody]")
        .addCriteria("mmm:hasBody/rdf:value", "panda")
        .addCriteria("mmm:hasTarget/mmm:hasSelector/rdf:value",
            "200,300", Comparison.ENDS_WITH);

List<PartMMM> result = qs.execute(PartMMM.class);
```

Line 2 of Listing 7 defines the namespace "**http://www.mico-project.eu/ns/mmm/2.0/schema#**" with the abbreviation "**mmm**" that is needed in order to use the following query criteria (because they will write "`mmm:`" instead of the whole namespace). After that, in Lines 3 through 5, different criteria are defined and added to the `QueryService`. The first criteria (satisfying requirement number two in the preceding enumeration) in Line 3 is used in order to define the RDF type of the associated body node (which in terms defines the "type" of the whole Part annotation). Therefore, the LDPath expression constitutes "hops" from the root over the edge `mmm:hasBody` to the body node and then does a type

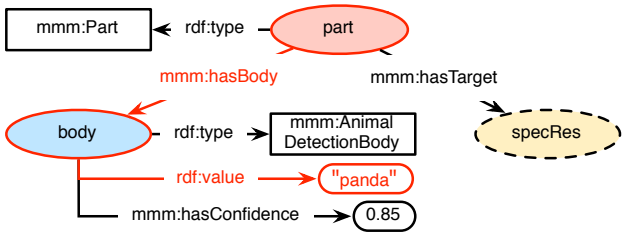
Figure 13 Graph-based visualisation of the LDPath expression "`mmm:hasBody[is-a mmm:AnimalDetectionBody]`".



check for the node type (relationship `rdf:type`) to be a `mmm:AnimalDetectionBody`. Figure 13 shows a visualisation of the path.

Line 4 of Listing 7 satisfies the requirement of looking for pandas. Therefore, the path to the body is again defined by the first "hop", then the value of its `rdf:value` relationship is compared against the String "panda". In this case, only a full String equality is tested. Figure 14 shows a graph visualisation. There are possibilities for other String comparisons, namely `contains`, `starts with`, and `ends with`. The basic behaviour (when no comparison operator is supported as third parameter) is set to exact match.

Figure 14 Graph-based visualisation of the LDPath expression "`mmm:hasBody/rdf:value`".



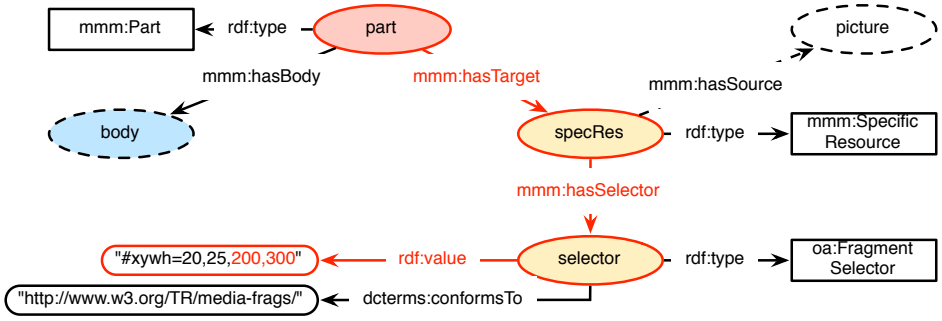
Requirement 4 is targeted at the fragment of the picture, so the path is directed to the target side of the given Part. It tests, if the supported fragment of the respective selector does match the size of 200 pixels width to 300 pixels height. Therefore, three "hops" in the graph are required (namely `mmm:hasTarget`, `mmm:hasSelector`, and `rdf:value`), in order to read the respective supported fragment. By using a String `ends with`-comparison, the last two parameters of the fragment (for width and height respectively) are compared. Those have to match the defined query. Figure 15 shows the graph visualisation for the last criteria.

The last requirement - to query for part annotations only - is satisfied by the execution call in Line 7, as the parameter for the starting point to query is set to `PartMMM.class` in the `.execute()`-method.

4.1.4 Anno4j Extended Features

The basic functionality of the ORM library Anno4j in order to query and persist RDF via Java POJOs has been covered in Section 4.1.1 and Section 4.1.3. The following sections will describe features that have been implemented for convenience purposes or to integrate a richer RDF feature support.

Figure 15 Graph-based visualisation of the LDPath expression "`mmm:hasTarget/mmm:hasSelector/rdf:value`", which is compared to end with the String "200,300".



The additions are namely: transactional behaviour, subgraphs and contexts, plugin extensibility, and input/output functionality.

- **Transaction Support**

  The Anno4j library features a transactional behaviour, enabling a user to create sets of actions, that either are done completely (**commit**) or not at all (**rollback**). The set becomes atomic. This enables the database to be consistent at any time, a possible client crash in the middle of its work procedure does not create an unclean or untraceable state of data. The basic behaviour (if no `Transaction` object is used) is set to auto-commit, so every action is persisted at the respective database automatically. A transaction itself has to be **started** (method `.begin()`) and then **commited** (method `.commit()`) or **rolled back** (method `.rollback()`). Listing 8 shows an example, that creates, begins, and ends a transaction while showing the possibility to create objects and a QueryService.

Listing 8: Use of a `Transaction` in Anno4j.

```
Anno4j anno4j = new Anno4j();

Transaction transaction = anno4j.createTransaction();
transaction.begin();

// Create and query using the Transaction object
ItemMMM item = transaction.createObject(ItemMMM.class);
QueryService qs = transaction.createQueryService();

transaction.commit(); / transaction.rollback();
```

- **Subgraphs and Contexts**

  A convenient feature of RDF is the use of contexts, that allow users to split their whole RDF graph into smaller contextualised subgraphs. Therefore, RDF triples are turned into so called **quads**, which have a fourth component after subject, predicate, and object that implements the URI of the subgraph the triple is to be contained in.

In Anno4j, the context can be utilised in one of two ways:

- Anno4j instance level: Two out of the four possible methods to create an object (`createObject( ... )`) support an URI parameter standing for the context. Creating an object this way will insert it in the respective subgraph.
- Transaction level (see Section 4.1.4 for transactions): Every Transaction object supports a `setAllContexts(String uri)` method, which defines the subgraph that the transaction is to write to and read from.

Listing 9 shows two examples using a specific context. Line 1 defines a new URI for a respective subgraph, while Line 4 creates an `ItemMMM` object in that subgraph. Line 7 creates a `Transaction` object and Line 8 changes its context to the defined context `uri`.

Listing 9: Setting a context for an Anno4j and Transaction object.

```
URI uri = new URIImpl("http://www.somePage.com/");

// Create an Item in the uri context
ItemMMM item = anno4j.createObject(ItemMMM.class, (Resource)
    uri);

// Create a Transaction object and define its context to uri
Transaction transaction = anno4j.createTransaction();
transaction.setAllContexts(uri);
```

- **Plugin Extensions**

  A plugin in the Anno4j context means the addition of an LDPath function in combination with underlying query logic. The logic will be evaluated at the point of time a query is executed. Because of this, the size of the result can be confined beforehand, rather than picking out the desired single entries afterwards on a bigger result set. In order to implement a plugin, the user has to define the LDPath function expression (`QueryExtension`), as well as the querying logic (`TestEvaluator`). An exemplary expression (taken from SPARQL-MM[KSK15]) can be seen in Listing 10. Integrating that criteria could lead to a result set of only those annotations, that detected both an elephant and a lion standing next to each other, the elephant found to the left side of the lion.

Listing 10: Exemplary plugin expression in an LDPath criteria.

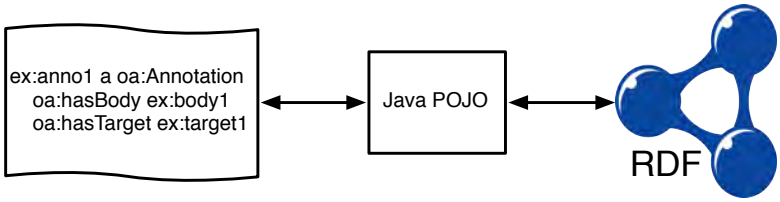
```
QueryService qs = anno4j.createQueryService();

qs.addCriteria("sparqlmm:leftBesides(\"elephant\", \"lion\")");
```

- **Input and Output**

  To improve the usability of the library, a small extension to the ORM has been implemented. Users can **parse** their RDF triples formulated in various RDF serialisations to create the respective Java objects, as well as **write** their Java objects as serialised RDF triples. Among the available serialisations are rdf/xml, ntriples, turtle, n3, jsonld, rdf/json, etc. So the mapping is “extended”, a textual component is added:

Figure 16 Extension to the ORM mapping of Anno4j.



In order to read a given RDF annotation (available as Java String), an `ObjectParser` object is needed. Its `.parse(String annotation, String uri, RDFFormat format)` requires the **annotation** as String, a **uri** for namespacing, and the supported **format**. It will then return a Java List of all parsed Annotations. Important to note is that all RDF nodes that are to be parsed need to be supported as respective Anno4j interfaces.

The `ObjectParser` holds all parsed annotations in a local temporary memory store. In order to persist them to your respective database, they have to be read from the parser and be persisted "by hand" then. Listing 11 shows an example reading a simple turtle annotation with a (empty) body and target associated.

Listing 11: Reading an annotation from RDF a given turtle serialisation.

```
String TURTLE = "@prefix oa: <http://www.w3.org/ns/oa#> ." +
            "@prefix ex: <http://www.example.com/ns#> ." +

            "ex:anno1 a oa:Annotation ;" +
            "   oa:hasBody ex:body1 ;" +
            "   oa:hasTarget ex:target1 .";

URL url = new URL("http://example.com/");

ObjectParser objectParser = new ObjectParser();
List<Annotation> annotations = objectParser.parse(TURTLE, url,
    RDFFormat.TURTLE);

objectParser.shutdown();
```

To write a given Anno4j Java object to respective RDF serialisations, the `ResourceObject` interface (which every Anno4j object descends from) supports a `.getTriples(RDFFormat format)` method, which returns the representation of the object as a Java String in the supported **format**. Listing 12 shows an example that writes a given item as turtle triples.

Listing 12: Writing a given Java item as turtle RDF serialisation.

```
ItemMMM item = anno4j.createObject(ItemMMM.class);
...
String itemAsTurtle = item.getTriples(RDFFormat.TURTLE);
```

4.2 Anno4CPP - C++ Proxy for Anno4j

While the usage of anno4cpp has been described already in Section 2.6.7, this one is about the underlying system making the Java anno4j framework available for the MICO C++ API and the MICO extractors as shown in Figure 1.

In order to automatically create C++ proxy classes representing all Java classes required to do annotations, we use a modified version of a tool called "JNIPP" (Original version: `https://github.com/mo22/jnipp`, MICO modified version: `https://bitbucket.org/mico-project/jnipp`). The tool receives a list of Java classes and generates C++ class proxies that use JNI as inner implementation. We extended "JNIPP" such that it is able to translate Java packages into C++ namespaces which avoids complicate class name mangling in the generated C++ classes. In order to retrieve the list of classes to be used, we developed a tool called "javadeps" (`https://bitbucket.org/mico-project/javadeps`) which takes a Java jar as input and generates a list of classes this input jar depends on. For the creation of such a jar an application was implemented called "'anno4jdependencies"'using all required "anno4j" and "Sesame" classes (`https://bitbucket.org/mico-project/anno4cpp`) and applied "javadeps" to it. The standard Java classes have also been added manually to the list. The C++ code generated by "JNIPP" out of the "anno4jdependencies.jar" is compiled into a static library and linked to the MICO C++ Persistence API which loads the JVM with the "anno4jdependencies.jar" and then can use the code through the proxies.

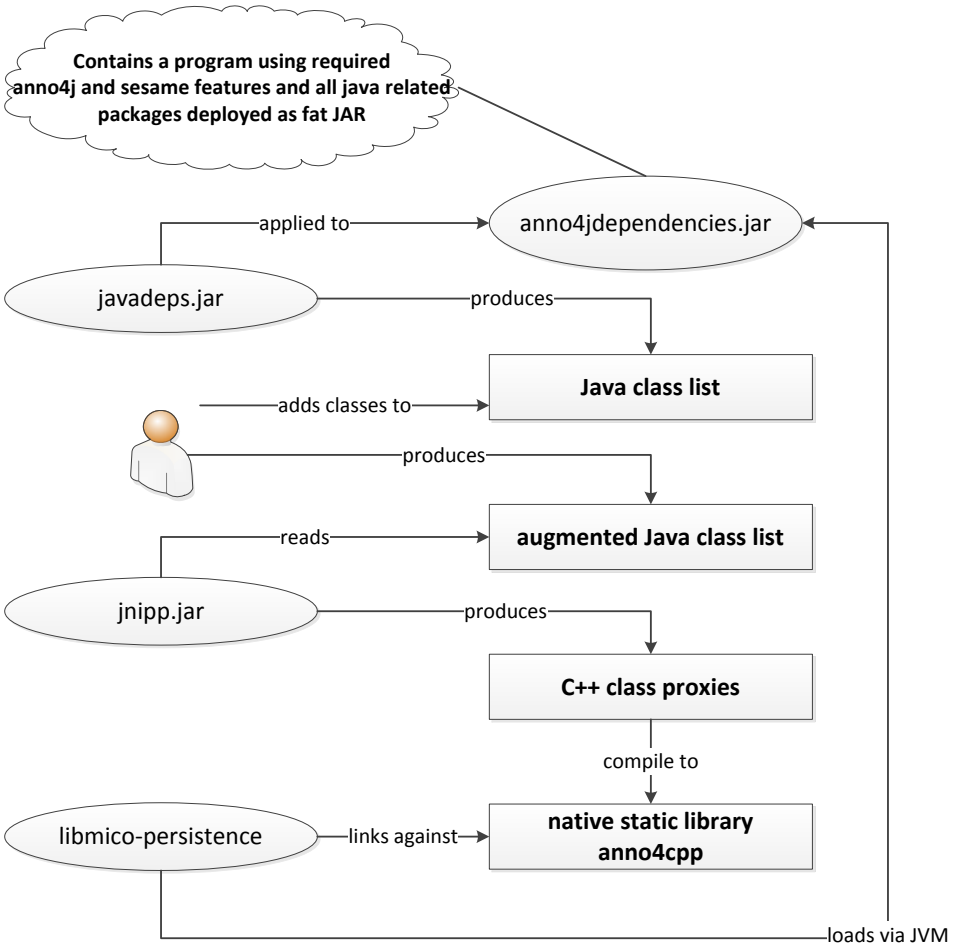
The whole generation process is managed by the anno4cpp project which takes care of fetching the generation tools, creating the proxies and producing the static library. The C++ MICO platform API build environment triggers the anno4cpp generation process during the build and takes care of using the correct version of the Java packages (e.g. mmm-anno4j, anno4j, etc.) by evaluating the Java API dependencies. Figure 17 shows a diagram of the process.

4.3 MMM Extension - Extractor Model

In the course of the evolution of the MICO Broker (see Section 3), the necessity for an extension to the MICO Metadata Model arose in order to incorporate more detailed information about the extractors of MICO workflows. Initial ideas have been documented in previous reports. An extractor is connected to its corresponding Part via the `oa:serializedBy` relationship. The supported information is aligned with the broker model depicted in Figure 5, which should be considered for details. The following RDF classes, relationships, and properties have been introduced to the MMM (all of the share the same namespace `"http://www.mico-project.eu/ns/mmm/2.0/schema#"`):

- **Extractor:** Class for an extractor in the MICO universe. An extractor is associated with exactly one `hasName`, `hasVersion`, and `hasStringId` property, and can have multiple `Mode` nodes attached via the `hasMode` relationship.
- **hasName:** The (MICO) name of the associated node.
- **hasVersion:** The (numerical) version of the associated node.
- **hasStringId:** The (uniquely identifiable) Id of the associated node, represented as String.
- **hasMode:** The relationship between a `Extractor` and its 0 to multiple `Mode` nodes.
- **Mode:** The mode of a given extractor. An extractor can have different modes that are distinguished by means of their parameters and IO data. A mode is associated with exactly

Figure 17 Overview of the proxy generation process for anno4cpp.



one hasConfigSchemaURI, hasOutputSchemaURI, hasStringId, and hasDescription property, and has possibly multiple Input, Output, and Param nodes attached via the relationships hasInputData, hasOutputData, and hasParam respectively.

- **hasConfigSchemaURI:** The URI where the config schema is to be found.
- **hasOutputSchemaURI:** The URI where the output schema is to be found.
- **hasDescription:** The (textual) description of the associated extractor detail.
- **hasInputData:** The relationship between a Mode and its associated Input node. One mode can have 0 to multiple hasInputData relationships associated.
- **hasOutputData:** The relationship between a Mode and its associated Output node. One mode can have 0 to multiple hasOutputData relationships associated.
- **hasParam:** The relationship between a Mode and its associated Param node. One mode can have 0 to multiple hasParam relationships associated.
- **IOData:** Superclass for the Input and Output classes. Every IOData node has exactly one hasIndex and hasCmdLineSwitch property associated. Additionally, it can have 0 to multiple hasMimeType, hasSemanticType, and hasSyntacticType relationships added.
- **hasIndex:** The (numerical) index for the associated node.
- **hasCmdLineSwitch:** The command line switch for the associated node.
- **hasMimeType:** The relationship between a IOData node and a given MimeType node. There can be 0 to multiple hasMimeType relationships.
- **hasSemanticType:** The relationship between a IOData node and a given SemanticType node. There can be 0 to multiple hasSemanticType relationships.
- **hasSyntacticType:** The relationship between a IOData node and a given SyntacticType node. There can be 0 to multiple hasSyntacticType relationships.
- **Input:** Subclass of IOData, representing an input requirement.
- **Output:** Subclass of IOData, representing an output requirement.
- **MimeType:** Class represents a mime type for a given IOData node. Has a hasFormatConversionURI and hasStringId property associated.
- **hasFormatConversionURI:** The URI that links to a defined format conversion.
- **SemanticType:** Class represents a semantic type of a given IOData node. Has exactly one hasName, hasDescription, and hasSemanticTypeURI property associated.
- **hasSemanticTypeURI:** The URI linking to the given semantic type.
- **SyntacticType:** Class represents a syntactic type of a given IOData node. Has exactly one hasAnnotationConversionSchemaURI, hasDescription, and hasSyntacticTypeURI property associated. Syntactic types can also link to 0 to multiple mime types via the hasMimeType relationship.

- **hasAnnotationConversionSchemaURI:** The URI that links to the annotation conversion schema.
- **hasSyntacticTypeURI:** The URI linking to the syntactic type.
- **Param:** Class represents a parameter of a given `Mode` node. A parameter has exactly one `rdf:value` and `hasName` property associated.

Figure 18 shows an example for the extractor vocabulary, illustrating a MICO audio demux extractor. For the sake of clarity, some nodes were left out (e.g. the `rdf:type` relationships, as the nodes are named equivalently, and for the example less important properties like conversion URIs). The extractor has two modes, `mode1` and `mode2`, which can have different settings, but in general deal with a similar task. In this example, the sampling frequency for the resulting audio file can differ (depending on what parameter has been defined by the extractor creator). `Mode2` is defined to take different video files as input, and produces audio as output, with a defined frequency of 16000.

Next to the whole extractor specific additions to the MMM, we also added an additional relationship `mmm:serializedWith` for semantical correctness, which connects a given `Part` node to the **Mode** of an extractor, that was responsible for its extraction. The edge is also included in Figure 18.

- **mmm:serializedWith:** The relationship between a `Part` and the `Mode` that was responsible for its extraction process. There can only be exactly one `mmm:serializedWith` relationship.

4.4 Conclusion and Outlook

This section contains a collection of lessons learned that have been gathered for the work done in WP3. They are related to different topics of WP3 and are discussed in short and in regards to their applicability to the current status of the MICO project and beyond.

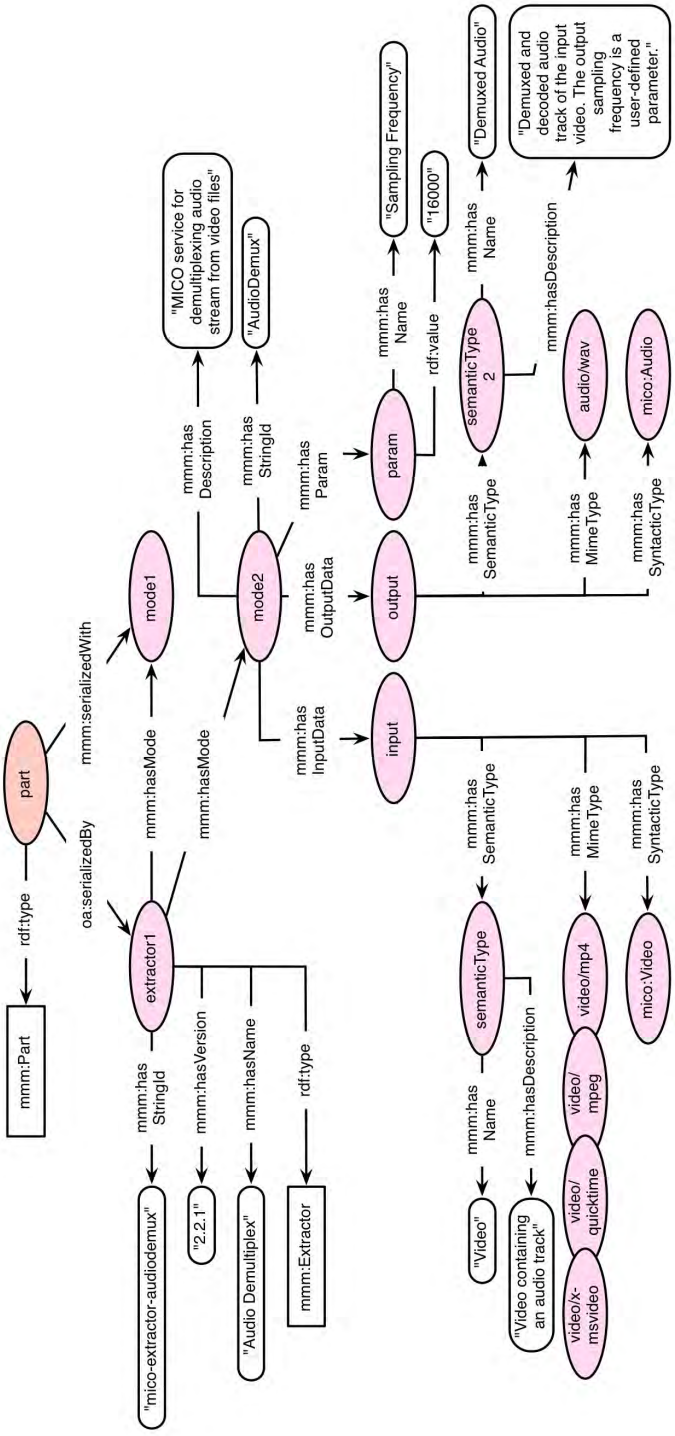
4.4.1 RDF Schema Enhanced Object-RDF-Mapping

The creation of Anno4j classes requires time. Additionally, at the current state, it is not possible to get an RDF schema out of the created classes. An "RDF Schema is a semantic extension of RDF. It provides mechanisms for describing groups of related resources and the relationships between these resources. RDF Schema is written in RDF [...]. These resources are used to determine characteristics of other resources, such as the domains and ranges of properties.", as described in [BG14].

Both the creation of RDF Schema out of existing Anno4j classes as well as the other direction – the generation of Anno4j classes out of existing RDF Schemata – are interesting approaches. To automatically create a schema allows easier integration of implemented ontologies for third parties, as the schema already includes a lot of crucial information of how to use the classes, relationships, and properties defined in the ontology. At the current state, the RDF schema is created by hand, which is deemed to be faulty.

The possibility of creating Anno4j classes out of RDF schemata enhances the base idea of Anno4j, which is to “connect” non-RDF and non-SPARQL experts to the Semantic Web. By allowing the integration of RDF Schema-based creation, the other side of this idea gets included: users who are already familiar with RDF, who might have already created ontologies and maybe schemata, can easily integrate their knowledge into the Anno4j concept. Consequently, their ontology can be made accessible to the non-RDF experts.

Figure 18 Exemplary RDF output for an audio demux extractor.



4.4.2 Validation of Created Metadata

This point is closely related to the one taken in Section 4.4.1. The invocation of criteria defined in RDF or OWL schemata could lead to a validation of the objects created by Anno4j classes and structures. In general, the validation of triples is a crucial point and is done via the implementation of an RDF schema.

At the current state, the only way to do validation in Anno4j is implementing the behaviour yourself via the Partial implementations (see Section 4.1.2), which might be a cumbersome task. An envisioned possibility for this task is to implement the desired RDF or OWL schema restrictions directly via Java annotations. These could be checked at point of creation of the respective Anno4j object or a whole transaction. Warnings or errors could then be communicated and handled accordingly. As an example, consider Listing 13, which introduces the Java annotation `@OWLMaxCardinality(x)`. Before according Anno4j instances are persisted, the specified OWL annotations are checked for their conformance.

Listing 13: Exemplary Anno4j interface implementing an envisioned OWL annotation

```
@Iri("http://example.org/schema#Child")
public interface Child extends ResourceObject {

        @OWLMaxCardinality(2)
        @Iri("http://example.org/schema#hasParent")
        void setParents(Set<Parent> parents);

        @OWLMaxCardinality(2)
        @Iri("http://example.org/schema#hasParent")
        Set<Parent> getParents();
}
```

4.4.3 Visualisation of Queried RDF Results

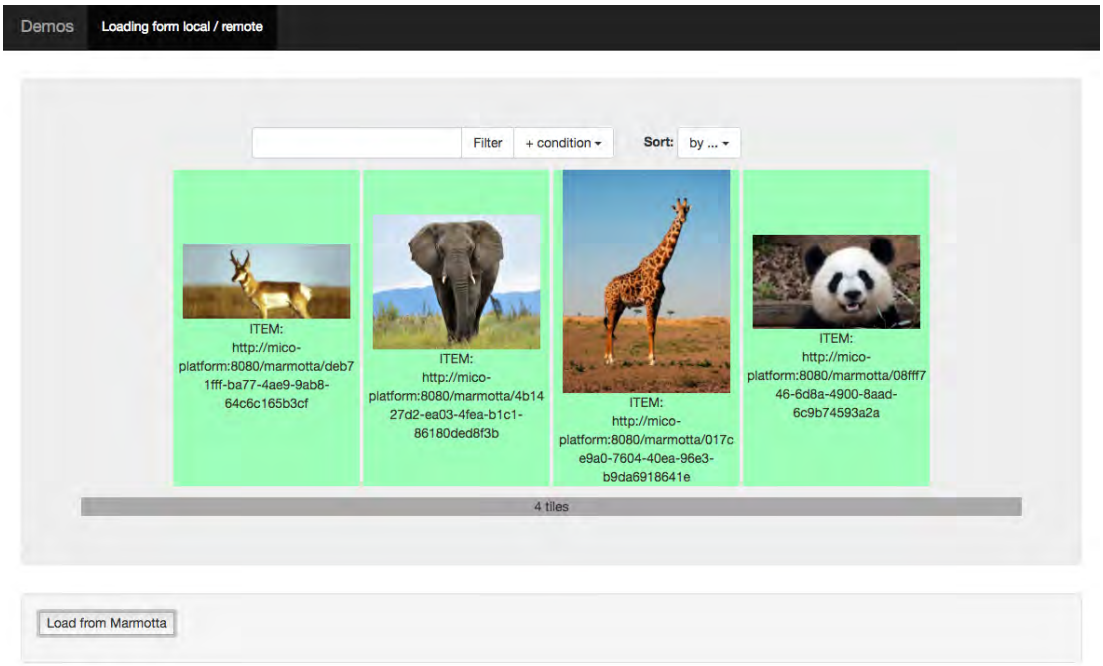
One of Anno4j's core contributions is to bring the technologies of the Semantic Web closer to those who are not yet familiar with mainly RDF and SPARQL. However, as persistence and querying is covered for the most part, there exists another component or barrier which could be difficult for non-Semantic Web experts: actually using or analysing the data that they create and request, as the data generally is presented in the form of classic RDF triples.

One possibility to solve this problem is to visualisations that let the user interact and explore the data on her own. Supporting graphs and, whenever possible, the underlying multimedia data in combination with its respective annotations could bridge the mentioned shortcoming. There are two approaches that are in progress right now.

The first approach is an extension to the Baloon Synopsis[23] implementation (see also [Sch+14b], [Sch+14a]) and is mainly used for demo purposes. Synopsis is a JQuery plugin and supports a node-centric RDF viewer designed in a modern tile design. As it is implemented in a modular way, filters and new views can easily be integrated in order to adapt the viewer to the MICO use case with Items, Parts, and Annotations. Figure 19 shows a screenshot of Synopsis, adapted to the MICO use case of an animal detection. Aggregated information can already be displayed at higher layers of the MMM graph, making it much easier for the user to step through the graph and understand the contained semantics. In this case, the picture that an extraction workflow has worked on is directly displayed in addition to the item URI. The plugin uses different colours for the tiles according to their information closeness. Every

[23] `http://schlegel.github.io/balloon/balloon-synopsis.html`

Figure 19 Screenshot of the Baloon Synopsis Plugin, adapted to animal detection use case, showing item query level



tile represents an RDF triple and is clickable, leading to a query of the respective triple. This way, the user can navigate through the presented graph.

Figure 20 shows the Synopsis result after the user has clicked on a given item to get further information about that item. In this case, the item contains the analysis results of the animal detection extractor, which would be contained in the body node of the part. This information is aggregated at part level in order to facilitate the navigation through the RDF once more.

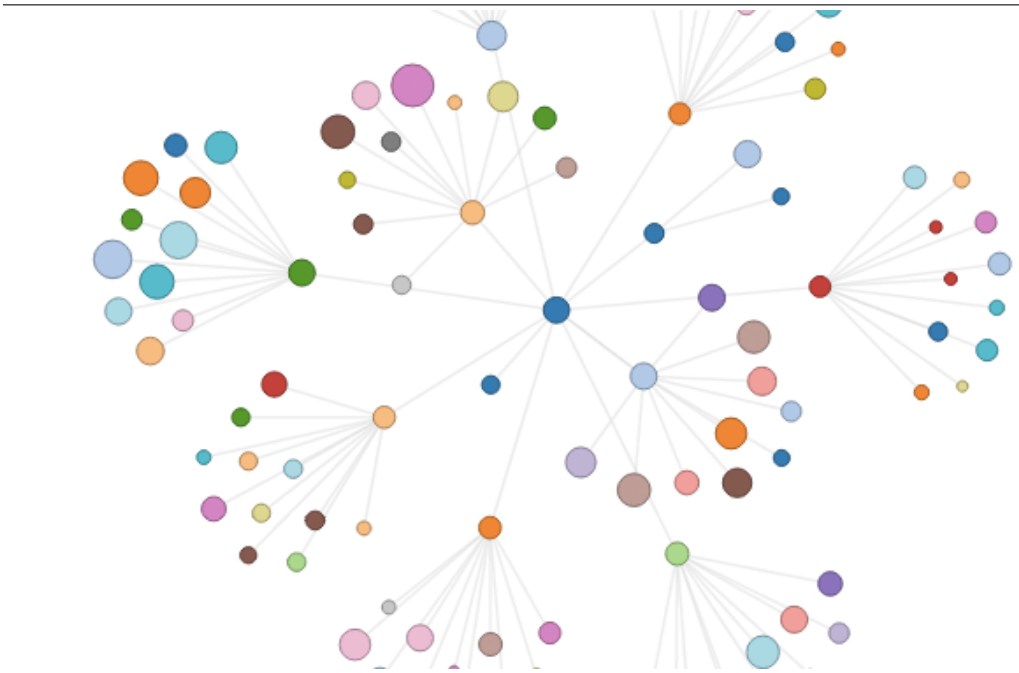
The second approach is realised as a bachelor's thesis at the University of Passau and should be considered as a more thorough approach, as it is designed directly for the MICO use case. However it will not be finished during the period of the MICO project. Initial ideas and thoughts are to be presented here. The approach is inspired by the explorative ideas of mindmaps. Mindmaps are a way of structuring ideas and thoughts in a tree-like or graph-like fashion. Oftentimes, these get big and extensive quite fast, so a good way of exploring them is needed. Some implementations only allow to view a part of the graph at a time, by clicking different nodes and points in the graph, the view is adapted. This idea is perfectly applicable for RDF graphs and especially the structure that is created by the MICO Metadata Model. With a starting point of the Item at top level, Parts and respectively their Annotations are to be explored "by hand" by the user. Figure 21 shows a first idea (displaying a D3[24] supported visualisation), of how the RDF graphs could be visualised. After selecting one item (which is then representing the node in the center), all of its Parts are arranged around it. After selecting a Part or group of Parts, the view changes and displays its content in a more detailed fashion. At this point, possibilities emerge that can further enhance the degree of information of an Annotation. Especially in the MICO use case, the information

[24] https://d3js.org/

Figure 20 Screenshot of the Baloon Synopsis Plugin, adapted to animal detection use case, showing part query level

that is extracted out of multimedia data can be integrated into the visualisation and its node-centric display.

Figure 21 Exemplary D3 RDF graph visualisation. Picture adapted from `http://flowingdata.com/2012/08/02/how-to-make-an-interactive-network-visualization/`.



5 Enabling Technology Modules for Cross-media Querying

The MICO project is about extracting information of various kinds from a broad range of media types like images, video, sound, and text, which is elaborated exhaustively in Section 2. This information is represented in a unified data model, which is based on W3C recommendation efforts around web annotations, which is described in Section 4. To retrieve parts of such information that satisfy use case specific needs users require a well defined toolkit for query formalization. In this report we describe three main contributions we did within the project and give hints and examples how to use the technology:

SPARQL-MM Extensions

In the past specification reports (Volume 2, Volume 4) we focused on Multimedia Information Retrieval, namely the SPARQL extension SAPRQL-MM [KSK15], which adds special relation-, aggregation- and accessor-functions for spatio-temporal media fragments to the de-facto standard query language for RDF. This extension has been lately released in version 2.0[25]. In this report we provide a compressed but exhaustive function list, outline the new features explicitly and give some usage examples.

Linked Data Information Retrieval

In some cases SPARQL is not the best solution for all use cases due to its complexity and steep learning curve. Especially when projects try to hide such complexity in order to be attractive to a broad community there is a need of solutions that offer a good tradeoff between feature completeness and simplicity in usage. This was the main reason for Anno4J (described in section 4) to integrate LDPath, a simple path-based query language, similar to XPath [CD99] or SPARQL Property Paths [HS13]. In this report we exhaustively introduce this query language, which we intentionally developed for querying and retrieving resources from the Linked Data Cloud. We outline the theoretical foundations of LDPath, define an exact Syntax Definition and give usage examples. Furthermore we give an overview on other projects that utilize LDPath in order to emphasize its versatility.

Semantic Media Similarity

As a third contribution we describe in this report a novel approach for media fragment similarity and give examples how this technology can be used. Even if this is a very early stage of the work it gives a good impression of the ongoing research and development in order to bring the web of data and multimedia even closer together.

5.1 SPARQL-MM Extensions

As described in former MICO Technical Reports, SPARQL-MM is a Multimedia Extension for SPARQL 1.1. Currently there are two implementations. One is based on Sesame (a higher level RDF API) and the other is integrated in Apache Marmotta and is optimized by using native features of the KiWi Triplestore and the underlying Postgres database. Both implementations have the same feature range. The features have been specified in report V2 and V4 and are explained in detail in report V3. In this section we do some recap and explain new features of the language version 2.0. In order to provide a smooth entry we start with a simple 2-steps tutorial. As in former reports we skip the SPARQL Prefix section for the matter of readability. All namespaces can be

[25] https://github.com/tkurz/sparql-mm/releases/tag/sparql-mm-2.0

looked up on http://prefix.cc. Note, that for SPARQL-MM 2.0 functions we use the prefix mm: <http://linkedmultimedia.org/sparql-mm/ns/2.0.0/function#>.

Step 1: Add dependency

The main reference implementation of SPARQL-MM is based on Sesame and thus uses SPARQL function extension mechanism. It is an in-memory implementation and not made for huge datasets but good for using it in smaller use case environments. The release is provided for open usage in Maven Central http://repo1.maven.org/maven2/com/github/tkurz/sparql-mm/2.0/. This enables you as a user to include the lib via several dependency frameworks. In case of using Maven add the following lines to your pom.xml:

```
<dependency>
        <groupId>com.github.tkurz</groupId>
        <artifactId>sparql-mm</artifactId>
        <version>2.0</version>
</dependency>
```

The functions are loaded via Java Service Loader, so you do not have to add it explicitly but can use it out of the box.

Step 2: Use it

This example assumes that you already added Sesame dependencies to your project. The example shows a simple test class that uses SPARQL-MM. If you run the example the duration of the media fragment (5.0 in this case) is printed on the command line.

```
public class SPARQLMMTest {

  public static void main(String [] args) throws Exception {

    String data = "<http:example.org/s> <http://example.org/p>
       <http://example.org/o#t=5,10> .";

    String query = "PREFIX mm:
       <http://linkedmultimedia.org/sparql-mm/ns/2.0.0/function#>" +
                   "SELECT ?duration WHERE {" +
                   "   ?s ?p ?o. " +
                   "   BIND(mm:duration(?o) AS ?duration)" +
                   "}";

    //init repository
    Repository repository = new SailRepository(new MemoryStore());
    repository.initialize();

    //import data
    RepositoryConnection connection = repository.getConnection();
    connection.add(new StringReader(data),"http://example.org/",
       RDFFormat.TURTLE);
```

```
    connection.commit();

    //query
    TupleQueryResult res =
        connection.prepareTupleQuery(QueryLanguage.SPARQL,
        query).evaluate();
    System.out.println(res.next().getBinding("duration")
            .getValue().stringValue());

    //close repository
    connection.close();
    repository.shutDown();
  }
}
```

SPARQL-MM 2.0 feature set

All existing features are listed in the feature matrix in Tables 12, 11 and 13. The shortname for parameter types are **S**patial Entity, **T**emporal Entity, Spatial-Temporal Entity (**ST**), **P**oint, **STR**ing, **B**oolean, and **D**ecimal. * defines a wildcard and + means one or more instances of a specific type are accepted as parameter. A detailed description of all features can be found on `https://github.com/tkurz/sparql-mm/blob/master/ns/2.0.0/function/index.md`. In this section we especially focus on the new features that are included by evaluating the lessons learned. **Extended Accessors**
In SPARQL-MM 2.0 we added several accessor functions that allow the usage of information hidden in Media Fragments. Now users can get information on shapes and timestamps, like duration, start, end, area and center point. Additionally the language now allows surface checks, e.g if a value is a media fragment, a fragment URI or if it includes spatial and/or temporal parts. The following query gets the fragment with the largest area and for a specific image. Thereby the system selects all fragments `?f` on an image, sorts it by area and selects the largest one.

```
SELECT ?fragment WHERE {
        ex:image ma:hasFragment ?f
        BIND (mm:area AS ?area)
}
ORDER BY DESC(?area)
LIMIT 1
```

Name Conflicts cleanup

In the former SPARQL-MM function the naming of several functions has been missleading. Namely there have been conflicts in the `equals`, `contains` and `overlaps` function because they are used for spatial, temporal and combined queries. This conflicts have been fixed by using a consistent naming schema, whereby clashing spatial functions are prefixed with *spatial* and temporal functions by *temporal*.

The Pixel-Percent Issue

When using pixel and percentage as dimensional units in parallel there have been calculation problems in version 1.0. More precisely relation functions failed when using different units for evaluation and produced false negatives. To overcome these problems we introduce pixel-to-percent and percent-to-pixel conversion functions. In the listed example we align the units before using well known SPARQL-MM functions. Note, for a proper alignment the width and height of a image has to be materialized in the data space as pixel values.

```
SELECT DISTINCT ?x WHERE {
        ?x ma:hasFragment ?f1, ?f2.
        ?x ma:width ?w.
        ?x ma:height ?h.
        BIND (mm:toPixel(?f1,?w,?h) AS ?pf1)
        BIND (mm:toPixel(?f2,?w,?h) AS ?pf2)
        FILTER mm:spatialEquals(?pf1,?pf2)
        FILTER(?f1 != ?f2)
}
```

Table 11 SPARQL-MM Aggregation Functions

Type	Name	Url	Params	Return
Spatial	boundingBox	mm:spatialBoundingBox	S+	S
	intersection	mm:spatialIntersection	S,S	S
Temporal	boundingBox	mm:temporalBoundingBox	T+	S
	intersection	mm:temporalIntersection	T,T	T
	intermediate	mm:intermediate	T,T	T
General	boundingBox	mm:boundingBox	ST+	ST
	interstion	mm:intersection	ST,ST	ST

Table 12 SPARQL-MM Relation Functions

Type	Name	Url	Params	Return
Spatial:Topological	equals	mm:spatialEquals	S,S	B
	disjoint	mm:disjoint	S,S	B
	touches	mm:touches	S,S	B
	contains	mm:spatialContains	S,S	B
	covers	mm:covers	S,S	B
	intersetcs	mm:intersects	S,S	B
	within	mm:within	S,S	B
	coveredBy	mm:coveredBy	S,S	B
	crosses	mm:crosses	S,S	B
	overlaps	mm:spatialOverlaps	S,S	B
Spatial:Directional	leftBeside	mm:leftBeside	S,S	B
	rightBeside	mm:rightBeside	S,S	B
	above	mm:above	S,S	B
	below	mm:below	S,S	B
	leftAbove	mm:leftAbove	S,S	B
	rightAbove	mm:rightAbove	S,S	B
	leftBelow	mm:leftBelow	S,S	B
	rightBelow	mm:rightBelow	S,S	B
Temporal	preceeds	mm:preceeds	T,T	B
	meets	mm:meets	T,T	B
	overlaps	mm:temporalOverlaps	T,T	B
	finishedBy	mm:finishedBy	T,T	B
	contains	mm:temporalContains	T,T	B
	starts	mm:starts	T,T	B
	equals	mm:temporalEquals	T,T	B
	ends	mm:ends	T,T	B
	startedBy	mm:startedBy	T,T	B
	during	mm:during	T,T	B
	finishes	mm:finishes	T,T	B
	overlapedBy	mm:overlapedBy	T,T	B
	metBy	mm:metBy	T,T	B
General	equals	mm:equals	ST,ST	B
	overlaps	mm:overlaps	ST,ST	B
	contains	mm:contains	ST,ST	B

Table 13 SPARQL-MM Accessor Functions

Type	Name	Url	Params	Return
Spatial	spatialFragment	mm:spatialFragment	*	STR
	hasSpatialFragment	mm:hasSpatialFragment	*	B
	area	mm:area	S	D
	hight	mm:hight	S	D
	width	mm:width	S	D
	xy	mm:xy	S	P
	center	mm:center	S	P
Temporal	temporalFragment	mm:temporalFragment	*	STR
	hasTemporalFragment	mm:hasTemporalFragment	*	B
	duration	mm:duration	T	D
	start	mm:start	T	D
	end	mm:end	T	D
General	mediaFragment	mm:mediaFragment	*	STR
	isMediaFragment	mm:isMediaFragment	*	B
	isMediaFragmentURI	mm:isMediaFragmentURI	*	B
	toPixel	mm:toPixel	ST	ST
	toPercent	mm:toPercent	ST	ST

5.2 Linked Data Information Retrieval

As described in Del. 4.1.1 there a two major streams of Semantic Web Query Languages. The first stream handles the Semantic Web as a set of triples (or quadruples in some cases) and have with SPARQL their major representative. The second stream focuses on the graph character of the Semantic Web, which is why its representatives are called RDF Graph Traversal Languages. Similar to graph database query languages like Cypher[26] or Gremlin [MD13], they start from base nodes (or resources) and use the outgoing links to compose query results while traversing the Web. This second stream is still in its beginnings but convinces by two major advantages:

database transparency: Graph Traversal Languages transparently support query federation over well known and widely used standards like HTTP.

problem proximity: Graph Traversal Languages follow the native structure of the Web (namely resources with links in between), which makes the usage intuitive and simple.

Both but mainly the problem proximity qualifies these languages for bridging the gap between complex Semantic Web data representation and the comprehension of pure Web developers. In the following sections we present LDPath as a representative of Graph Traversal Languages that we used, adapted and formalized within the MICO project.

5.2.1 Theoretical Foundations

In this Section we describe the formal semantics of LDPath, which are fundamental for a proper definition of the query language but may be skipped by readers who just want to get an overview on the basic concepts, which also are described by example in Section 5.2.2. As basis we use a simple triple based model for representing the data that is distributed over the *Web of Linked Data*. In particular, we assume that the web consists of interlinked documents containing RDF triples. The formal definition of the Web of Linked Data is based on [HF12].

Definition 1
For the scope of this document, we let

- $\mathscr{I}$ be the set of all possible identifiers (e.g. URIs, IRIs),
- $\mathscr{L}$ be the set of all possible constant literals (e.g. strings, natural numbers, etc..),
- $\mathscr{R} = \mathscr{I} \cup \mathscr{L}$, and
- t be a data triple $t \in (\mathscr{I} \times \mathscr{I} \times (\mathscr{I} \cup \mathscr{L}))$

We model the *Web of Linked Data* as a structure of interlinked documents. These documents are accessed via their identifiers $d \in \mathscr{D}$ and contain the data that is represented as a set of data triples.

Definition 2
The **Web of Linked Data** $\mathscr{W}$ is a tuple $(\mathscr{D}, data, adoc)$ where:

- $\mathscr{D}$ is the set of symbols that represent LD documents, $\mathscr{D} \subseteq \mathscr{I}$
- $data : \mathscr{D} \to 2^{\mathscr{I} \times \mathscr{I} \times (\mathscr{I} \cup \mathscr{L})}$ is a total mapping such that $data(d)$ is finite for all $d \in \mathscr{D}$

[26] `http://neo4j.com/docs/stable/cypher-introduction.html`

- $adoc : \mathscr{I} \rightarrow \mathscr{D}$ is a partial, surjective mapping.

Put in simple words *data* is the function to retrieve the *data* (i.e. the RDF description) of a document d from the Web, while *adoc* is used to resolve the *associated document d* for a given identifier $i \in \mathscr{I}$.

LDPath works on two primary constructs, *selectors* and *filters*. Selectors are used to navigate the Web of Linked Data, while filters reduce the result set based on specific conditions.

Definition 3
For the following definitions of selectors and filters (see Section 5.2.1), we let

- $\mathscr{S}$ be the set of all Selectors, and
- $\mathscr{F}$ be the set of all Filters.

Selectors

In this section, the set of available selectors is defined. The reference implementation in Apache Marmotta supports additional selectors, however these put additional constraints on the data and work in a "best effort" base. Some examples are listed in Section 5.2.3.

Definition 4
The **PropertySelector** is the basic "navigation" from one resource r to a direct neighbour, linked with the given property p. It is defined as a function $s^{\text{Prop}} : 2^{\mathscr{I} \cup \mathscr{L}} \times \mathscr{I} \rightarrow 2^{\mathscr{I} \cup \mathscr{L}}$:

$$s^{\text{Prop}}(R, p) = \{o \mid r \in R \cap \mathscr{I}; \langle r, p, o \rangle \in data(adoc(r))\}$$

Definition 5
The **WildcardSelector** is a relaxed PropertySelector, following all available links from the given start resource r. It is defined as a function $s^{\text{WC}} : 2^{\mathscr{I} \cup \mathscr{L}} \rightarrow 2^{\mathscr{I} \cup \mathscr{L}}$:

$$s^{\text{WC}}(R) = \{o \mid r \in R \cap \mathscr{I}; \exists p \in \mathscr{I} \Rightarrow \langle r, p, o \rangle \in data(adoc(r))\}$$

Definition 6
By concatenating two Selectors, it is possible to traverse a complex path through the web of Linked Data. This is called **PathSelector** and defined as a function $s^{\text{Path}} : 2^{\mathscr{I} \cup \mathscr{L}} \times \mathscr{S} \times \mathscr{S} \rightarrow 2^{\mathscr{I} \cup \mathscr{L}}$:

$$s^{\text{Path}}(R, s_1, s_2) = R \circ s_1 \circ s_2 = s_2(s_1(R, .), .)$$

Definition 7
The **UnionSelector** merges the results of two other selectors. It is defined as a function $s^{\cup} : 2^{\mathscr{I} \cup \mathscr{L}} \times \mathscr{S} \times \mathscr{S} \rightarrow 2^{\mathscr{I} \cup \mathscr{L}}$:

$$s^{\cup}(R, s_1, s_2) = s_1(R, .) \cup s_2(R, .)$$

Definition 8
The **IntersectSelector** reduces the results of two other selectors to their intersection. It is defined as a function $s^{\cap} : 2^{\mathscr{I} \cup \mathscr{L}} \times \mathscr{S} \times \mathscr{S} \rightarrow 2^{\mathscr{I} \cup \mathscr{L}}$:

$$s^{\cap}(R, s_1, s_2) = s_1(R, .) \cap s_2(R, .)$$

Definition 9
The **RecursiveSelector** applies a selector multiple times. It is defined as a function $s^{\text{Rec}} : 2^{\mathscr{I}\cup\mathscr{L}} \times \mathscr{S} \times \mathbb{N}_0 \times \mathbb{N}_0 \to 2^{\mathscr{I}\cup\mathscr{L}}$:

$$s^{\text{Rec}}(R,s,l,u) = \begin{cases} R & \text{if } l = 0, u = 0 \\ R \cup R' & \text{if } l = 0, u > 0 \\ R' & \text{if } l > 0, u > l \\ \emptyset & \text{otherwise} \end{cases}$$

where $R' = s^{\text{Rec}}(s(R,.),s,\max(l-1,0),\max(u-1,0))$.

There are two special versions of this selector:

- the **StarSelector** is defined as $s^*(R,s) = s^{\text{Rec}}(R,s,0,\infty)$,
- the **PlusSelector** is defined as $s^+(R,s) = s^{\text{Rec}}(R,s,1,\infty)$

These two can cause problems with termination if $\mathscr{W}$ is infinite. Also a circle detection must be added when implementing those selectors, otherwise the evaluation might not terminate even for finite $\mathscr{W}$. The implementation in Apache Marmotta (see Section 5.2.4) is able to detect circles and stops the execution accordingly.

Definition 10
The **TestSelector** applies a filter (see 5.2.1) to the result set of a selector. It is defined as a function $s^{\text{Test}} : 2^{\mathscr{I}\cup\mathscr{L}} \times \mathscr{S} \times \mathscr{F} \to 2^{\mathscr{I}\cup\mathscr{L}}$:

$$s^{\text{Test}}(R,s,f) = f(s(R,.),.)$$

Filters

Using filters it is possible to filter the result set $R \subseteq (\mathscr{I}\cup\mathscr{L})$ of a selector function to a subset that fulfills certain conditions. In this section we will describe and define the most common filters. However, the implementation in Apache Marmotta (see Section 5.2.4) supports additional filters that add extra restrictions on the data. Examples of such filters will be shown in Section 5.2.3.

Definition 11
A **LiteralTypeFilter** allows to select only literals of a certain literal type. It is defined as a function $f^{\text{LitType}} : 2^{\mathscr{I}\cup\mathscr{L}} \times \mathscr{I} \to 2^{\mathscr{I}\cup\mathscr{L}}$:

$$f^{\text{LitType}}(R,t) = \{l \mid l = \langle l^v, l^t\rangle \in R \cap \mathscr{L}; l^t = t\}$$

To simplify the definition of LDPath, we assume that *language tags* of literals can be expressed through special *literal types*. For the scope of this document, we say that e.g. the language tag 'en' can be expressed as `@en` ⇔ `^^rdf:langString$en`.

Definition 12
A **PredicateFilter** checks the simple presence of a predicate/path starting from a resource. It is defined as a function $f^{\text{Has}} : 2^{\mathscr{I}\cup\mathscr{L}} \times \mathscr{S} \to 2^{\mathscr{I}\cup\mathscr{L}}$:

$$f^{\text{Has}}(R,s) = \{r \mid r \in R; s(\{r\},.) \neq \emptyset\}$$

In other words, it checks that the given selector evaluates to something other than $\emptyset$ when applied to the resource to check.

Definition 13
A **PropertyFilter** checks the the given selector evaluates to a specific value. It is defined as a function $f^{\text{Is}} : 2^{\mathscr{I}\cup\mathscr{L}} \times \mathscr{S} \times \mathscr{I} \cup \mathscr{L} \to 2^{\mathscr{I}\cup\mathscr{L}}$:

$$f^{\text{Is}}(R,s,v) = \{r \mid r \in R; v \in s(\{r\},.)\}$$

Definition 14
An **IntersectionFilter** allows the combination of two other filters, taking the intersection of both their result sets. It is defined as a function $f^{\text{And}} : 2^{\mathscr{I}\cup\mathscr{L}} \times \mathscr{F} \times \mathscr{F} \to 2^{\mathscr{I}\cup\mathscr{L}}$:

$$f^{\text{And}}(R,f_1,f_2) = f_1(R,.) \cap f_2(R,.)$$

Definition 15
An **UnionFilter** allows the combination of two other filters, taking the union of both their result sets. It is defined as a function $f^{\text{Or}} : 2^{\mathscr{I}\cup\mathscr{L}} \times \mathscr{F} \times \mathscr{F} \to 2^{\mathscr{I}\cup\mathscr{L}}$:

$$f^{\text{Or}}(R,f_1,f_2) = f_1(R,.) \cup f_2(R,.)$$

5.2.2 Syntax Definition

In the following part, we will describe the syntax used to define LDPath rules and programs. An LDPath Program is a set of rules, optionally with a label.

Please note that the syntax defined here is a simplified version since the actual implementation also supports constructs that are formally incomplete or not decidable. Also, some simplifications have been applied to make the syntax grammar more readable. The full syntax specification can be found in the repository of the implementation[27].

⟨*program*⟩ ::= ⟨*prefix*⟩* ⟨*rule*⟩+

⟨*prefix*⟩ ::= '`@prefix`' ⟨*lname*⟩ '`:`' ⟨*iri*⟩ '`;`'

⟨*rule*⟩ ::= (⟨*lname*⟩ '`=`')? ⟨*selector*⟩ '`;`'

⟨*selector*⟩ ::= ⟨*atomicSelector*⟩
| ⟨*compoundSelector*⟩
| ⟨*testingSelector*⟩

⟨*atomicSelector*⟩ ::= ⟨*property*⟩ *// PropertySelector*
| '`*`' *// WildcardSelector*
| '`(`' ⟨*selector*⟩ '`)`'

⟨*compoundSelector*⟩ ::= ⟨*pathSelector*⟩
| ⟨*Selector*⟩ '`&`' ⟨*Selector*⟩ *// IntersectSelector*
| ⟨*Selector*⟩ '`|`' ⟨*Selector*⟩ *// UnionSelector*
| '`(`' ⟨*selector*⟩ '`)`' ⟨*bounds*⟩ *// RecursiveSelector*

⟨*pathSelector*⟩ ::= ⟨*atomicSelector*⟩ ('`/`' ⟨*pathSelector*⟩)? *// PathSelector*

[27] `http://s.apache.org/rR`

⟨*testingSelector*⟩ ::= ⟨*atomicSelector*⟩ '`[`' ⟨*test*⟩ '`]`' *// TestSelector*

⟨*test*⟩ ::= ⟨*atomicTest*⟩
| ⟨*test*⟩ '`&`' ⟨*test*⟩ *// IntersectionFilter*
| ⟨*test*⟩ '`|`' ⟨*test*⟩ *// UnionFilter*

⟨*atomicTest*⟩ ::= '`^^`' ⟨*property*⟩ *// LiteralTypeFilter*
| '`@`' ⟨*languageTag*⟩ *// LiteralTypeFilter (language)*
| ⟨*selector*⟩ *// PredicateFilter*
| ⟨*selector*⟩ '`is`' ⟨*value*⟩ *// PropertyFilter*
| '`(`' ⟨*test*⟩ '`)`'

⟨*property*⟩ ::= ⟨*lname*⟩ '`:`' ⟨*lname*⟩ | '`<`' ⟨*iri*⟩ '`>`'

⟨*value*⟩ ::= ⟨*property*⟩ | '`"`' ⟨*string*⟩ '`"`'

⟨*iri*⟩ ::= IRI as in RFC3987 [DS05]

⟨*languageTag*⟩ ::= Tag as in RFC5646 [PD09] | '`none`'

⟨*lname*⟩ ::= [A-Za-z0-9_]+

⟨*bounds*⟩ ::= '`{`' ⟨*number*⟩ '`,`' ⟨*number*⟩ '`}`' | '`*`' | '`+`'

The grammar covers the set of formal definitions described in section 5.2.1. To give a better understanding of LDPath we give some example queries that are executed on a small test set of RDF resources outlined in Listing 14. We refrain from using MICO specific vocabularies (e.g. the Annotation Model) in order to keep the example data simple and best fitting for the main goal, which is the presentation of language features. The set contains resources from several data sources including complex databases (e.g. dbpedia) and simple files (e.g. a FOAF file). As start resource we take the FOAF profile of a fictive person *John Doe* (`ex:john`). For the matter of compactness we do not instance one example per each definition but combine several constructs within the queries. For the same reason we skip the prefix section.

Listing 14: Sample Data in Turtle syntax.

```
@prefix ex: <http://example.com/data/> .
@prefix foaf: <http://xmlns.com/foaf/0.1/> .
@prefix schema: <http://schema.org/> .
@prefix db: <http://dbpedia.org/resource/> .
@prefix dbo: <http://dbpedia.org/ontology/> .
@prefix skos: <http://www.w3.org/2004/02/skos/core#> .

  ex:doe foaf:firstname "John" ;
  foaf:surname "Doe" ;
  foaf:birthday "12-31" ;
  ex:married_to ex:mary ;
  foaf:knows <http://www.wikier.org/foaf#wikier> ;
  dbo:birthPlace db:London .
```

```
ex:mary a schema:Person ;
schema:name "Mary Doe" ;
ex:has_job ex:elementary_teaching .

ex:elementary_teaching a skos:Concept ;
skos:broader ex:teaching .
ex:teaching a skos:Concept ;
skos:broader ex:education .
ex:education a skos:Concept ;
skos:topConceptOf ex:job_sectors .
```

As mentioned, LDPath query evaluation starts on one or more specific resources. The following examples always start with one specific resource, namely `ex:doe`, which represents John.

Example 1: What's your surname and when are you born, John?

This is basic path query that uses the PropertySelector from Def. 4. The result is a map.

```
surname  = foaf:surname;
birthday = foaf:birthday;

Result: {surname:["Doe"], birthday:["12-31"]}
```

Example 2: Who are your contacts, John?

This more complex query shows how LDPath supports the combination of various ontologies. It uses the UnionSelector (Def. 7) and the UnionFilter (Def. 15). In addition we do not care on the relation and use a WildcardSelector (Def. 5).

```
friends = *[foaf:Person|schema:Person]
/ (foaf:name|schema:name);

Result: {friends:["Mary Doe","Sergio Fern%*\'a*)ndez"]}
```

It has to be mentioned that the IntersectSelector (Def. 8) and the IntersectionFilter (Def. 14) works analogous to the intersections so we do not outline them in separate examples.

Example 3: How is your home country called in Germany, John?

This example shows how LDPath allows to transparently query various datasets. To choose the language of the result literal we use the LiteralTypeFilter (Def. 11).

```
geburtsland = dbo:birthPlace
/ dbo:country
/ rdfs:label[@de];

Result: {geburtsland:["Vereinigtes Königreich"]}
```

Example 4: In which sector does your wife work, John?

For this query we want to query for the top concept of *Mary Doe's* employment. Therefore we use the RecursiveSelector (Def. 9) and the PredicateFilter (Def. 12). The query uses the SKOS hierarchy and collects all broader related concepts if they match the filter (so only a TopConcept is added).

```
sector = (
ex:maried_to / (skos:broader)*
)[skos:topConceptOf];

Result: {sector:[ex:education]}
```

5.2.3 Extensions

While the LDPath constructs described in Section 5.2.1 are theoretical sound and complete, many use-cases require more expressive language constructs even if it cannot be guaranteed that they always evaluate correctly with respect to the open-world assumption. However, some of those constructs are described in this section.

Selector- and Filter-Extensions

Definition 16
The **ReversePropertySelector** is the inverse version of the PropertySelector. It tries to follow *incoming* links. It would be defined as a function $s^{\text{RevProp}} : 2^{\mathscr{I}\cup\mathscr{L}} \times \mathscr{I} \to 2^{\mathscr{I}\cup\mathscr{L}}$:

$$s^{\text{RevProp}}(R,p) = \{s \mid \langle s,p,r\rangle \in data(adoc(s)); s \in \mathscr{I}; r \in R\}$$

Since the Web generally only contains *forward* links, this selector would require the complete data corpus rendering the distributed architecture of the web void. However, for some special use-cases the above restrictions do not apply additional constraints (e.g. in a closed world scenario) the ReversePropertySelector can be useful.

Definition 17
A *NotFilter* would negate the regarding filter. It would be defined as a function $f^{\text{Not}} : 2^{\mathscr{I}\cup\mathscr{L}} \times \mathscr{F} \to 2^{\mathscr{I}\cup\mathscr{L}}$:

$$f^{\text{Not}}(R,f) = R \setminus f(R,.)$$

Transformers

Originally introduced for the use-case of "Semantic Search" (see Section 5.2.5), transformers proved to be useful in other scenarios too.

After evaluating an LDPath statement, the result is passed to the transformer and applies a final operation to the resources in the result set. A transformer is defined as a function $t : 2^{\mathscr{I}\cup\mathscr{L}} \to 2^X$, where X solely depends on the implementation of the transformer. As an example, the transformer `xsd:double` in listing 15 is defined as $t^{\text{xsd:double}} : 2^{\mathscr{I}\cup\mathscr{L}} \to 2^{\mathbb{R}}$.

The reference implementation in Apache Marmotta (see Section 5.2.4) ships with a set of transformers for the most common XSD-Datatypes that try to compensate for incomplete data. E.g, literals that come without an explicit type set will be cast to the result type, if that fails the value is removed from the result set.

Listing 15: LDPath example with transformer

```
name = foaf:surname :: xsd:string;
birthday = foaf:birthday :: xsd:date;
weight = bio:weight :: xsd:double;
```

So even if the original data comes without type definition, the result of the LDPath statement will be typed.

Functions

Also functions were originally introduced for the "Semantic Search" scenario (see Section 5.2.5).

Internally, a function is handled like a selector but allows more complex operations and can be dynamically parameterized. A function in LDPath is defined as $\mathfrak{f} : 2^{\mathscr{I} \cup \mathscr{L}} \times A \rightarrow 2^{\mathscr{I} \cup \mathscr{L}}$ where $a \in A$ is a list of selectors as arguments for the function $\mathfrak{f}$: $a =< a_1, \ldots, a_n >, a_i \in \mathscr{S}$.

The argument selectors are evaluated within the context of the function, their results passed as input to the function. A simple example illustrates this for the function `fn:concat(.)`:

Listing 16: `fn:concat` example

```
full_name = fn:concat(foaf:givenname, " ", foaf:surname);

Result: {name = ["John Doe"]}
```

5.2.4 Implementations

LDPath is currently implemented for Java by Apache Marmotta[28] and relies on a number of backends, such as Apache Jena models, OpenRDF Sesame repositories or simple files on disk. As described above, the language can be extended with new functions, which can be easily added by implementing a Java interface (and using ServiceLoader[29] for its registration).

LDPath is also exposed in Marmotta as a simple JSON based RESTful protocol[30], which is being implemented by Redlink on its cloud platform[31]. On the client side currently there is support for LDPath Java, PHP, Python and Javacript.

5.2.5 Uses Cases

LDPath simplifies the access to Linked Data resources. Therefore, behind the main aim of querying the Linked Data Cloud, LDPath enables a broader range of use cases. In the MICO project, the language is mainly used and well supported by Anno4J, like described in Section 4. But there are more scenarios where LDPath plays a central role, which are worth to be mentioned here.

Listing 17: LDPath templating example.

```
<@namespace foaf="http://xmlns.com/foaf/0.1/" />
<@namespace schema="http://schema.org/" />
<html>
  <head>
```

[28] http://marmotta.apache.org/ldpath
[29] https://docs.oracle.com/javase/6/docs/api/java/util/ServiceLoader.html
[30] http://marmotta.apache.org/platform/ldpath-module#protocol
[31] http://dev.redlink.io/api/1.0-BETA/#ldpath-get

```
    <title>
      Friends of
      <@ldpath path="(foaf:name|schema:name)::xsd:string"/>
    </title>
  </head>
  <body>
    <h1>
      Friends of
      <@ldpath path="(foaf:name|schema:name)::xsd:string"/>
    </h1>
    <ul>
      <@ldpath path="*[foaf:Person|schema:Person]">
        <li>
          <@ldpath path="(foaf:name|schema:name)::xsd:string"/>
        </li>
      </@ldpath>
    </ul>
  </body>
</html>
```

Templating: The first is LDPath as templating language[32]. LDPath implements an extension of the FreeMarker template engine that allows constructing templates with LDPath statements for inserting and iterating over resources' values. Listing 17 shows a basic example how can it be used inside FreeMarker.

Semantic Search: Since LDPath provides a very convenient way to flat Linked Data complex paths, it is a straight-forward way to interface with NoSQL stores such as Apache Solr or ElasticSearch. The Linked Media Framework [KSB11] uses LDPath to configure the access to the RDF data for providing Semantic Search[33]. The resources are automatically indexed in Apache Solr offering different perspectives on the data managed in a very performant way, like faceting, pivoting, etc. Using the same example data as for former sections, Listing 18 outlines a LDPath program for indexing friends of person in a flat document structure using de-normalization.

Listing 18: Semantic Search configuration example.

```
name = *[foaf:Person|schema:Person]
/(foaf:name|schema:name) :: xsd:string;
```

LDPath is also used in Apache Stanbol to flat resources for internal storage and special retrieval facilities[34].

LIDO: LIDO[35] (Linked Data Object Mapper for Java) allows easy access to Linked Data resources by annotating POJOs in a similar way to JPA. It is an experimental Open Source Project and follows a similar approach like Anno4J but without the focus on a specific kind of use case.

[32] http://marmotta.apache.org/ldpath/template
[33] https://bitbucket.org/srfgkmt/lmf/wiki/Module-Semantic-Search
[34] https://stanbol.apache.org/docs/trunk/components/contenthub/contenthub5min
[35] https://github.com/tkurz/lido

Listing 19: LIDO mapper example.

```
@Type("http://schema.org/Person")
public class Employee {

        @Path("<http://schema.org/name>")
        public String name;

        @Path("<http://schema.org/description>")
        public LangString langString;

        @Path("<http://schema.org/weight>")
        public double weight;
}
```

These POJOs are stubbed and filled with data from remote resources using a LDPathMapper. This programming approach decouples the complexity of RDF graph structure and the object-oriented world of Java and allows a seamless integration of Linked Data resources in existing applications.

Listing 20: LIDO - retrieving data.

```
// create mapper for employees
LDPathMapper<Employee> employees  =
                                new LDPathMapper<>(dataClient,
                                    Employee.class);

// get one employee
URI e1 = new URI("http://example.org/e1");
Employee employee = employees.findOne(e1);
```

5.2.6 Future Work

In this section we presented the formal semantics of LDPath, a representative of Graph Traversal Languages. Furthermore we gave use case examples for the main LDPath concepts. LDPath as language is out there since quite a while now (first versions are date at the end of 2011[36]), but in its recommendation step it focused more on industrial use cases than in the research community. This lead to a broad user community of developers in the Apache Software Foundation, where projects such as Apache Marmotta and Apache Stanbol do use LDPath for accessing RDF datasets with different purposes. This facts led to the decision that LDPath is a suitable, stable, expressive and easy to use solution for Anno4J.

[36] https://lists.w3.org/Archives/Public/public-lod/2011Dec/0011.html

5.3 Semantic Media Similarity

In the former section we presented SPARQL-MM as an extension of the query language SPARQL to multimedia facilities. In order to define a feature set for the extension we analyzed over 70 query languages of the last three decades, which leads us to the feature set outlined in Table 14. In order to keep the report compact we skipped the generic query language features here (like *Transitive Closure*, *Relational Completeness*, etc) and focused on multimedia specific functionalities. In a survey we are currently working on we will do a exhaustive analysis including generic as well as specific requirements. As the reader can see, many of the features are already covered by SPARQL and/or existing extensions

Table 14 Requirements for Multimedia Query Languages

Title	Description	Satisfaction
Universal	Is suitable for multi-modal multimedia data	RDF, MA, Media Fragment URIs
Uncertainty	Supports the concept of partial truth	fSparql, SPARQL arithmetics
Spatial Operations	Provides Spatial Functions	SPARQL-MM
Temporal Operations	Provides Temporal Operations	SPARQL-MM
Evolution	Supports asset time series	×
Metadata Awareness	Support structural and content-description based metadata operations	SPARQL-MM, SPARQL, Fulltext Index
Media Similarity	Provide similarity metrics and functions	×
Weighting	Allow usage of relevance values for sorting and boosting	SPARQL, SPARQL arithmetic

(including SPARQL-MM). But there are still two major points missing which are evolution and media similarity. Whereby evolution is a very rarely used feature, media similarity is crucial for many use cases. Therefor we decided to focus this in our further work. In this section we give an brief overview on Semantic Similarity and sketch an approach of the integration of fragment relations to these. An extension of SPARQL-MM to Media Similarity is currently in progress but not yet finalized and this will be presented in the near future.

5.3.1 Semantic Media Similarity

The basic idea of our approach is the integration of Multimedia Fragments with well-known concept similarity measurements. Semantic concept similarity is a metric over a set of terms, whereby the idea of distance between them is based on the likeness of their meaning, like described in [Har+15]. There mainly exist two technological branches which are topological and statistical similarity. Topological Similarity is based on interlinked concept graphs and underlying ontologies. This technology is a good fit for RDF based systems as data and metadata is already provided as named graph. But in the later past (due to the dawn of the third iteration of neuronal networks and related machine learning) Satistical Similarity becomes more prominent. It is based on a text corpus and uses a vector space model for word correlation calculations. In order to be free in the choice of the concept similarity our approach is metric independent as long as the metric produces a normalized output, namely values between `0` and `1`, whereby `0` means no similarity and `1` states that two concepts are the same. Furthermore we proceed on the assumption of a linear distribution.

The overall algorithm is outlined in Listing 21. Note, that the algorithm performs a similarity query based on an image j and an image set I based on spatial fragments. In the preparation we identify the most important fragment f^* for every image i (lines 2-7) and calculate the relative positions of fragments of i regarding f^* (lines 9-11). This is the actual comparison part is located at lines 15 - 23. The Semantic Media Similarity value of two images i, j is the sum of the Semantic Similarity of their most significant fragments and the maximum Semantic Similarity for each f_i to every f_j. This sum is weighted, which means the contribution of a similarity value depends on the fragment significance and the fragment similarity *fragSim* (which is defined by spatial deviation).

Listing 21: Semantic Fragment Similarity.

```
1  prepare(I) {
2    foreach image i in I {
3      foreach fragment f_i of i {
4         calculate_fragment_significance(f_i)
5      }
6
7      f* = most_significant_fragment(i)
8
9      foreach fragment f_i of i {
10       calculate_relative_fragment_positions(f_i,f*)
11     }
12   }
13 }
14
15 query(j,I) {
16   foreach image i in I {
17     i.s = semSim(i.f.first, j.f.first)
18     foreach f_j in j.f.rest {
19       f_i = fragment_with_most_semantic_similarity(i.f.rest)
20       i.s += semSim(f_j, f_i) * significance(f_j) * fragSim(f_j,f_i)
21     }
22   }
23   order_by_similarity(I)
24 }
```

A clear formal definition of fragment significance and spatial deviation is the basis for a proper and efficient implementation. In the following we outline these definitions starting with Normalized Fragment Significance.

Normalized Fragment Significance

Normalized Fragment Significance denotes the prominence based on relative position (which we call normalized center distance) and the relative areas of the fragments (normalized area). The assumption is, that a fragment is more significant if it is bigger and position nearby the center.

Definition 18

The normalized center distance NCD for a fragment f_i regarding an image i is defined as

$$NCD(i, f_i) = \frac{|\vec{c}_{(i)} - \vec{c}_{f_{(i)}}|}{\sqrt{width(i)^2 + height(i)^2)}} \tag{1}$$

whereby c are center vectors.

Definition 19
The normalized area NA for a fragment f_i regarding an image i is defined as

$$NA(i, f_{(i)}) = \frac{width(f_{(i)}) * height(f_{(i)})}{width(i) * height(i)} \tag{2}$$

Definition 20
Normalized Fragment Significance NFS describes the significance of an image fragment f_i within an image i and is defined as

$$NFS(i, f_{(i)}) = NA(i, f_{(i)}) * (1 - NCD(i, f_{(i)})) \tag{3}$$

Semantic Fragment Similarity

Within the algorithm we use the most significant fragment which we define formally as:

Definition 21
Let $F_{(i)}$ be the set of all fragments of an image i. The most significant fragment $f^*(i) \in F_{(i)}$ is a fragment that fulfills the following condition:

$$NFS(i, f^*_{(i)}) >= f_{n_{(i)}} \quad \text{for} \quad f_{n_{(i)}} \in F_{(i)}, n \in \mathbb{N} \tag{4}$$

This allows the identification of the most significant fragment. It can be used to identify a fragment fixpoint fp for the calculation of similarity of fragments regarding relative positioning.

Definition 22
A fragment fixpoint $\vec{\phi}_{(i)}$ for an image i is defined as the center of the most significant fragment $f^*(i)$.

Definition 23
Let $f_{(i)}$ a fragment of image i. The relative center vector $\vec{\psi}_{f_{(i)}}$ is defined as the displacement of the center vector $\vec{c}_{f_{(i)}}$ of the fragment and the fragment fixpoint of i:

$$\vec{\psi}_{f_{(i)}} = \vec{\phi}_{(i)} - \vec{c}_{f_{(i)}} \tag{5}$$

With this the cosine similarity can be used to calculate the normalized spatial fragment similarity.

Definition 24
With the taken definitions the Normalized Spatial Fragment Similarity (NSFS) of the fragments $f_{(i)}, f_{(j)}$ of two images i, j regarding their fixpoints can be calculated as follows:

$$NSFS(f_{(i)}, f_{(j)}) = \frac{\vec{\psi}_{f_{(i)}} * \vec{\psi}_{f_{(j)}}}{\|\vec{\psi}_{f_{(i)}}\| * \|\vec{\psi}_{f_{(j)}}\|} \tag{6}$$

5.3.2 Further Work

In this section we gave a short introduction into a novel Semantic Media Similarity approach. As the reader can see, the approach is still a early work in progress and lacks an evaluation. Therefor a round of A-B Testing will presently be made with a group of 30 users. There we will test the approach in a controlled environment using a subset of the MS Coco (Common Objects in Context)[37] dataset of annotated images. Additionally we will extend the approach to temporal fragments.

[37] http://mscoco.org/

6 Enabling Technology Modules for Cross-media Recommendations

WP5 is about providing a framework that uses both collaborative filtering and content-based approaches for recommendations within the platform. The relevant use cases can be separated into two domains:

1. Recommendation use cases which require *collaborative filtering only*, e.g., evaluating user likes or site accesses.

2. *Cross-media recommendation* use cases, for which the problem may be defined as follows: *The task of finding a **suitable** selection of **media** items, where there are at least two different types of input media sharing the same context.*

As for the latter, following the ideas outlined in [Kö+16], we identified two relevant use cases in the project which utilize information created by the *extractors* of WP2, using the MICO infrastructure to provide cross-media recommendation:

a) The IO10 *Editor Support* Use Case, which uses MICO to match text analysis on videos with text analysis of journal articles, to support an editor in finding related archive content.

b) The Zooniverse *Chat Analysis*, which combines sentiment and content analysis within chats with animal detection. The goal is to identify *debated* content, for which regular Zooniverse users are unable to reach a consensus.

The main goal of this chapter is to describe these applications and their respective implementation, and it is structured as follows:

- Section 6.1 provides an overview over the recommendation approaches implemented within the MICO project.
- Section 6.2 discusses the location of the recommendation components within the overall MICO architecture, and provides an overview of the API functions that can be used by the showcases.
- Section 6.3 describes the components status, including brief usage instructions.
- Finally, Section 6.4 discusses the WP5 results by comparing them to the technology enablers defined in the initial version of this report (V3).

6.1 Recommendation Approaches

Recommendation in MICO focuses on two main data sources: media analysis results provided by the developed extractors (see Section 2) and user data provided by our showcase partners. Naturally, this leads to a distinction between collaborative filtering (CF) and content based recommender systems, the applications of which will be discussed in this section.

6.1.1 Collaborative Filtering

While not being a cross-media recommendation approach, using user behaviour to find similar items is a fundamental requirement to every recommendation approach. The following use cases required a pure CF-based approach:

Greenpeace Site Statistics

The Greenpeace magazine provides site usage data in csv format, containing the visited page and a user id, i.e. tuples like `/il-pesce-azzurro-nel-canale-di-sicilia/,view,21358`. While more detailed information (e.g., the amount of time a user spends reading a page) would have been beneficial, there are some overlaps between users which can be used to derive recommendations.

InsideOut10 provides a recommendation widget that can be used to present the recommendations in *Wordlift* on their GitHub page[38] mark (see Figure 22 for an example). The recommendation widget has the goal to drive traffic to other articles that are relevant for the user and related to the article itself. To overall goal is to keep readers on the website, increasing user engagement, and ultimately boosting the number of subscribers.

Zooniverse Image Likes

The Zooniverse project circles around *subjects* e.g., pictures of animals, that will be annotated and discussed by their users. A direct user-subject recommendation, where subjects will be recommended to users that are likely to have a certain annotation performance and will be motivated by this item is difficult. Finding psychological factors that increase a users time on the website is still ongoing research at Zooniverse [Bow+15]. Dispatching unannotated subjects does not meet the requirements of our showcase partner.

However, besides annotating subjects, users have the opportunity to flag subjects as *favorites*, for the Snapshot Serengeti, e.g., pictures of sunsets, an exceptional amount of animals or special events, like a predator hunting down another animal (see Figure 23 for examples). In such cases, we can assume that some users have a different taste of pictures and can recommend favorite images.

For implementing CF-based recommendations within the MICO project, Prediction.io was selected within the intial phase of the project. To handle the complex installation, a Prediction.io module was developed within a docker container which can be installed automatically, if CF-functionalities are required within the specific MICO instance. The engine template used within MICO is the universal recommender[39] - for installation instructions, see Section 6.3.
For both aforementioned CF use cases, MICO provides a recommendation endpoint returning a list of items for a given item. Prediction.io will be trained with an initial dataset on install, further user data (*events*) can be added later at any time. For the API endpoints, see the `/reco` endpoint in Table 15.

6.1.2 Content Based recommendation

Content-based recommendation include all recommender functionalities that exploit item metadata. For MICO, this is the most relevant source for cross-media recommendations, and it exploits metadata annotated by *extractors* from WP2.

Editor Support Use Case

The setting for the editor support case is as follows: A journal editor (e.g., for the Italian Greenpeace Magazine) is creating a new article. To create a rich user experience, she is advised to link

[38] https://github.com/wanbinkimoon/wordlift-playground
[39] https://github.com/pferrel/template-scala-parallel-universal-recommendation

related content from the journal's video archive, whose meta data is of poor quality. Instead of having to looking through the content all by herself, she can use MICO that analyses the archive and matches the content information extracted there with the content of the article currently written.

Getting information for the article uses the NER (named entity recognition) components of MICO.

The workflow for videos is more complex: The audio part of all videos is extracted and MICO's speech to text component is used to generate input for the NER.

It is assumed that the archive content is analysed a priori on a MICO instance. The analysis of the articles will be much faster, and can be repeated for different states of the writing process. The methods of the `ner/` and `video/` endpoint will be used to get matching MICO-items, i.e., videos from the archive. Additionally, this process can be complemented with results from the collaborative filtering.

Zooniverse Chat Analysis

As discussed in the introduction, recommending *subjects* to *users* is subtle. Even the initial idea of avoiding the annotation of (pre-classified) empty image does not seem to increase the user engagement. Therefore we decided against recommending certain subjects to certain users to improve classification.

Instead, the focus was switched to the administrative staff of a citizen science project. On Zooniverse, administrators have the possibility to forward selected images to experts (e.g., biologists), if they think that regular users are not able to give a precise annotation.

Of course, doing this large-scale contradicts the idea of a citizen science project (that is: relieve researchers from annotation tasks). So this forwarding is considered costly and should be avoided. Given the large amount of data, this can only done for a very selected set of items, and there should be criteria that help the administrative stuff in finding items where such forwarding is indeed beneficial.

MICO's animal detection and text analysis components allow to assess the *debatedness* of subjects, analysing both the image as well as the discussion for this image. This assessment can be used to automatically flag certain subjects for moderator attention.

The features that are currently used for calculating a debated score are:

- Sentiment analysis results

Figure 22 Wordlift recommendation widget for Greenpeace data

- Comparison of found animals between image and text
- Discussion length

Whether or not a certain subject is debated according to that definition can be queried using the `reco/zoo/{subject_id}/is_debated` endpoint (see Table 15).

6.2 Platform Integration

The location of recommendation components within the MICO platform is depicted in Figure 24. As for implementation, all API functions are part of the plaform's *showcase-webapp*. To save resources, the installation of Prediction.io is optional. See Section 6.3 for respective installing instructions. Following the MICO Technical Report Volume 4, the functional description of the main components is the following (numbers refer to Figure 24):

1 Recommendation engine including custom recommendation modules
Basic services which are potentially reusable for various recommendation use cases are implemented here. This includes, e.g., the code for the entity matching for the editor support use case.

2 Recommendation API
Code that is specific to a single user story / need will be put in separate modules that connect to the recommendation modules. While media analysis data is available via Anno4j, the recommendation API also offers capabilities to collect user data from the showcases, and forwards it e.g. to the Prediction.io event server for re-training the recommendation model.

The MICO platform exposes utility functions for recommendations as an API implemented within the *showcase-webapp* as REST web services. The API description is available as a *JSONdoc* file at `http://mico-project.bitbucket.org/api/rest/?url=reco.json`. For sake of completeness, the available endpoints are listed in Table 15.

3 Prediction.IO
As described in the previous reports, Prediction.io is responsible for machine-learning based collaborative filtering tasks. While Prediction.io offers a rich feature set, it comes with many dependencies. To simplify using it inside the platform, *docker* was used. For WP5, a custom dockerfile was written that allows to easily deploy Prediction.io with a single command and encapsulate its dependencies (see Section 6.3.1).

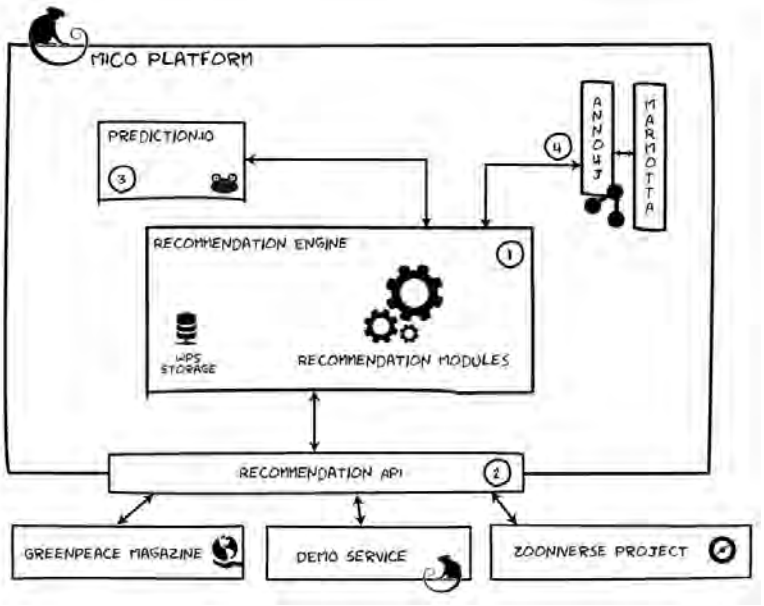
4 Marmotta & Anno4j
Data is stored inside the platform using Marmotta. WP5 will query this data using the Marmotta SPARQL endpoint, using the querying capabilities of Anno4j when applicable. Anno4j is

Figure 23 Examples of favourite images (subjects ASG001p8rj, ASG001s4b8, and ASG001s5my)

Table 15 API Overview.
Please see `http://mico-project.bitbucket.org/api/rest/?url=reco.json` for a detailed API documentation and all parameters. On a default configured MICO platform, the endpoints are available as a sub resource of `http://mico-platform:8080/showcase-webapp`.

Method	*URL*	*Description*
GET	/ner/{source}/transcript	Get speech-to-text results for given source
GET	/ner/{source}/entities	Get linked entities for given source
GET	/reco/pioevents	Outputs all events stored by the current prediction.io instance
GET	/reco/piostatus	Determines whether prediction.io is running on the system
GET	/reco/dockerstatus	Determines whether docker is running on the system
GET	/reco/piosimplereco	Returns recommendation for a given item using Prediction.io
POST	/reco/createentity	Adds new event to prediction.io
GET	/reco/zoo/{subject_id}/discussion/relatedsubjects	Returns related subjects for a given subject id
GET	/reco/zoo/{subject_id}/is_debated	Returns whether a given subject is debated by its users
GET	/videos/default	Get a list of default filenames
GET	/videos/entities/{entity}	Gets annotations for a given video
GET	/videos/related/{source}	Gets semantically related item for a given sourcename
GET	/videos/analyzed	Get a list of annotated videos

Figure 24 Overview of the MICO recommendation architecture.



discussed inside this report in chapter 4. Along with the development of the work package 5 components, the *MicoQueryHelper* functions were created, allowing an easy, reusable way to filter for certain content types. As an example, querying all analyzed (mp4-)video items is as simple as:

```
List<ItemMMM> analyzedItems =
    micoQueryHelper.getItemsByFormat("video/mp4")
```

Note that despite the query is run against a Marmotta SPARQL endpoint, the preparation of the query is transparent to the user and the returned ItemMMM classes provide all methods the data model describes.

Regarding implementation, the code is split on two repositories: a dedicated repository for the recommendation engine, and the code for the Recommendation API, which is part of the showcase-webapp within the platform. An overview of the repository structure is provided in Figure 25.

Figure 25 Recommendation repository structure

Platform https://bitbucket.org/mico-project/platform
- [...]
- showcase-webapp
  - src/main/java/eu/mico/platform
    - **reco** ..... Recommendation API Code
- [...]

Recommendation https://bitbucket.org/mico-project/recommendation
- Demo ..... Collection of WP5 working examples
- PredictionIO .... Code related to deployment and configuration of prediction.io inside docker
- RecoApi ..... Legacy API Code
- EditorSupport ..... Implementation of editor support use case

6.3 Recommendation Engine & Demo Code

6.3.1 Installing the recommendation API

The API is developed as a tomcat webapp inside the platform repository: `bitbucket.org/mico-project/platform/src/HEAD/showcase-webapp/`.

The deployment can be done automatically using maven, providing the respective MICO instance in the `pom.xml`. More details regarding the configuration can be found here: `https://bitbucket.org/snippets/mico-project/Adxrz`. Having built a *.war file, a manual upload to the tomcat manager webapp is possible as well: The showcase-webapp.war must be placed within the directory `/var/lib/tomcat7/webapps/`. The tomcat server detects and deploys the file automatically (startup usually takes two minutes).

6.3.2 Installing Prediction.io

To simplify using it inside the platform, *docker*[40]is used. For WP5, a custom dockerfile was written that allows to easily deploy Prediction.io with a single command and encapsulate its dependencies. After installing docker on the MICO platform (see next section), there are three ways to get a running docker container:

1. Start from a base ubuntu image and install Prediction.io there. See section "Prepare Prediction.io base image" for details.
2. Use the dockerfiles uploaded in the MICO Bitbucket repository. See section "Running Prediction.io using a prepared MICO Dockerfile" for details.
3. Use a prepared image from Dockerhub. See `https://hub.docker.com/r/thomasidmt/wp5/` for details.

Install Docker on platform
Open a shell inside the MICO platform and install Docker via apt and make it runnable as a non-root user:

```
$ sudo apt-get update
$ sudo apt-get install curl
$ curl -sSL https://get.docker.com/ | sh
$ sudo usermod -aG docker user
```

Logout, Login again

Prepare Prediction.io base image
Prediction.io will be installed on an ubuntu base image (at time of writing: Ubuntu 16.04.1 LTS - xenial). In mico-image:

```
$ docker run -t -i ubuntu
```

In container shell:

[40] `https://www.docker.com/what-docker`

```
$ apt-get update
$ apt-get install curl default-jdk python-pip
$ pip install pymongo
$ pip install predictionio

$ curl
    https://raw.githubusercontent.com/actionml/PredictionIO/master/bin/install.sh
    -o pio-install.sh

$ bash pio-install.sh
```

At the beginning of the installation process, answer the asked question as follows. Installation roughly takes 30 minutes.

- Installation path: /root/PredictionIO
- Vendor path: /root/PredictionIO/vendors
- Backend: Elasticsearch + HBASE
- Receive updates: No
- Distribution: Debian
- Install Java: No

In the success case, the output is as follows (please do not run those commands, yet):

```
Installation of PredictionIO 0.9.7-aml complete!
Please follow documentation at
    http://docs.prediction.io/start/download/ to download the engine
    template based on your needs

Command Line Usage Notes:
To start PredictionIO Event Server in the background, run: 'pio
    eventserver &'
To check the PredictionIO status, run: 'pio status'
To train/deploy engine, run: 'pio [train|deploy|...]' commands

Please report any problems to: support@prediction.io
```

Leave docker container:

```
$ exit
```

Get container ID in mico-image:

```
$ docker ps -all
```

The output looks like

```
CONTAINER ID    IMAGE     COMMAND        CREATED            STATUS
                          PORTS     NAMES
00e0d52c621a    ubuntu    "/bin/bash"    35 minutes ago     Exited (0)
    9 seconds ago              kickass_bartik
```

In this case, use `00e0d52c621a` or `kickass_bartik` as `<ID>`:

```
$ docker commit -m "installed pio 097-aml" -a "username" <ID>
    mico-aml:v1
```

Test Prediction.io status
Enter Docker container:

```
$ docker run -t -i mico-aml:v1
```

In Docker container:

```
$ /root/PredictionIO/bin/pio-start-all
$ /root/PredictionIO/bin/pio status
```

The last printed line is supposed to be `[INFO] [Console$] Your system is all ready to go.`

Running Prediction.io using a prepared MICO Dockerfile
Docker accepts git repositories instead of local files as references to Dockerfile & Co. This allows us to store the Dockerfile inside bitbucket for testing.

You have to adjust the path accordingly (e.g., change the username), see `https://docs.docker.com/reference/commandline/build/` for syntax details. During the *build* phase, all of the prediction.io requirements (Spark, elasticsearch...) are automatically downloaded and configured inside the docker container.

In a public release this will be published on a public repository, making this adjustment step only required for development.

```
$ docker build --tag="wp5"
    https://user@bitbucket.org/mico-project/\_
     recommendation.git#greenpeace:PredictionIO/docker
$ docker run -p 8000:8000 -p 7070:7070 -p 9000:9000 wp5
```

Engine is deployed to port 8000 and can be used by the demo interface or showcase partners. Check `http://localhost:8000` for the overview page.

6.3.3 Running the WP5 Demo application

The Mico WP5 demo is a node.js[41] application. It is available inside the recommendation bitbucket repository: `https://bitbucket.org/mico-project/recommendation/src/HEAD/Demo/`

To run the demo, install node.js (tested with version 6.9.1), npm, the package manager is included inside the package. Get the code from the repository and run the following commands:

[41] https://nodejs.org/en/

Table 16 Summary of the technology enablers implemented within this work package. Section 5.7 of report V4 (final specification) gives a more detailed description.

TE-501: User activity and context monitor	User data collection is responsibility of the showcase partners, user behaviour is stored within the Prediction.io backend.
TE-502: User similarity calculator	Covered by Prediction.io.
TE-503: Item Similarity calculator	Covered by the cross media entity matching and the *debated* classification for the chat analysis.
TE-504: Cross-Modal Content recommender	The components developed within WP5 as described in Section 6.2.

```
$ npm install
$ bower install
$ npm run-script start
```

As default, the demo is available at http://localhost:3000

6.4 Deviations from Report V3 - Enabling Technology Modules: Initial Version

The Technical Report V3 [Aic+14] focused mainly on the usage and deployment of Prediction.io-engines. While Prediction.io is still in use, the configuration and deployment of engines is now performed automatically by the MICO platform. Over the course of the project, and due to the change of priorities, focus somewhat shifted, and deviations from the originally planned Technology Enablers 501-504 were discussed in Section 5.7 of report V4 (final specification). Table 16 provides an informal summary on how the technology enablers were implemented.

6.5 MICO Recommender System – Summary and Outlook

The goal of WP5 was about providing a framework that provides both collaborative filtering and content-based approaches for recommendations. As outlined earlier, there were significant changes to the functionalities required by showcase partners, which resulted in a focus on the following use cases in y3:

- Editor support use case
- Debated item detection
- Article recommendation based on usage data
- Favorite image recommender

While these use cases are specific to showcase partners and the data they provide, the implemented recommendation modules are by no means limited to those applications: The provided API is generic and applicable to many potential adopters of the MICO platform with similar requirements for recommendation.

Due to the fact that WP5 has always depended on to (a) the existence and performance of relevant WP2 extractors and (b) the availability of the whole MICO platform, WP5 represents the technical

WP that is "last in the foodchain" of technical WPs, resulting in the fact that WP5 implementations started later than planned, and there was only limited time for implementing WP5 functionalities, and to experiment with them, especially on the cross-media recommendation use cases. As a possible lesson learned for the future, one could argue that this could have been at least partially avoided by decoupling WP5 activities from other WPs by creating and using mockup annotations. On the other hand, the chosen approach led to WP5 effectively validating crucial platform functionalities across all technical WPs, which was very valuable - many platform issues have been detected by WP5 exploitation all of them - extraction, extractor orchestration, and anno4j/querying.

Hence, while the recommendation functionalities provided by WP5 should be considered a first basic functional set, they are fully integrated in the platform, and they are very extensible. We believe that there is a huge overall potential in using recommendations based on CF and content-based approaches, and that the recommendation framework within MICO now provides a solid basis for further extensions in many directions, e.g. using semantic similarity metrics for the entity matching components.

References

[Aic+14] Patrick Aichroth et al. *D2.3.1/D3.3.1/D4.3.1/D5.3.1 Enabling Technology Modules: Initial Version*. Tech. rep. MICO, 2014.

[Aic+15] Patrick Aichroth et al. *Dx.2.2 Specifications and Models for Cross-Media Extraction, Metadata Publishing, Querying and Recommendations: Final Version*. Deliverable. MICO, Oct. 2015. URL: http://www.mico-project.eu/wp-content/uploads/2016/01/Dx.2.2-SPEC_final_READY_FOR_SUBMISSION.pdf.

[Ber+16] Emanuel Berndl et al. "Idiomatic Persistence and Querying for the W3C Web Annotation Data Model." In: *Joint Proceedings of the 4th International Workshop on Linked Media and the 3rd Developers Hackshop co-located with the 13th Extended Semantic Web Conference ESWC 2016, Heraklion, Crete, Greece, May 30, 2016*. Ed. by Raphaël Troncy et al. Vol. 1615. CEUR Workshop Proceedings. CEUR-WS.org, 2016. URL: http://ceur-ws.org/Vol-1615/limePaper5.pdf.

[BG14] Dan Brickley and R.V. Guha. *RDFS RDF Schema 1.1*. W3C Recommendation. http://www.w3.org/TR/rdf-schema/. W3C, Feb. 2014.

[Boa12] DCMI Usage Board. *DCMI Metadata Terms*. Tech. rep. Dublin Core Metadata Initiative, June 2012. URL: http://dublincore.org/documents/dcmi-terms/.

[Bow+15] Alex Bowyer et al. "This Image Intentionally Left Blank: Mundane Images Increase Citizen Science Participation." In: *Human Computation and Crowdsourcing: Works in Progress and Demonstrations. An Adjunct to the Proceedings of the Third AAAI Conference on Human Computation and Crowdsourcing*. 2015.

[CD99] James Clark and Steve DeRose. *XML Path Language (XPath), Version 1.0*. Recommendation. W3C, Nov. 1999. URL: http://www.w3.org/TR/xpath/.

[DS05] M. Duerst and M. Suignard. *RFC 3987: Internationalized Resource Identifiers (IRIs)*. RFC 3987 (Proposed Standard. Internet Engineering Task Force, Jan. 2005. URL: https://www.ietf.org/rfc/rfc3987.txt.

[Fel+10] P.F. Felzenszwalb et al. "Object Detection with Discriminatively Trained Part Based Models." In: *IEEE Pattern Analysis and Machine Intelligence* 32.9 (2010), pp. 1627–1645.

[Fou04] The Apache Software Foundation. *Apache Camel*. Oct. 2004-2015. URL: http://camel.apache.org/ (visited on 10/30/2015).

[Har+15] Sébastien Harispe et al. *Semantic Similarity from Natural Language and Ontology Analysis*. Synthesis Lectures on Human Language Technologies. Morgan & Claypool Publishers, 2015. DOI: 10.2200/S00639ED1V01Y201504HLT027. URL: http://dx.doi.org/10.2200/S00639ED1V01Y201504HLT027.

[HF12] Olaf Hartig and Johann-Christoph Freytag. "Foundations of traversal based query execution over linked data." In: *Proceedings of the 23rd ACM conference on Hypertext and social media*. ACM. 2012, pp. 43–52.

[HS13] Steve Harris and Andy Seaborne. *SPARQL 1.1 Query Language*. Recommendation. W3C, Mar. 2013. URL: http://www.w3.org/TR/sparql11-query/.

[KSB11] Thomas Kurz, Sebastian Schaffert, and Tobias Bürger. *LMF - A Framework for Linked Media*. Workshop on Multimedia on the Web (MMWeb2011) in conjunction with iSemantics2011. Sept. 2011.

[KSK15] Thomas Kurz, Kai Schlegel, and Harald Kosch. "Enabling access to Linked Media with SPARQL-MM." In: *Proceedings of the 24nd international conference on World Wide Web (WWW2015) companion (LIME15)*. 2015. DOI: 10.1145/2740908.2742914.

[Kö+16] Thomas Köllmer et al. "A Workflow for Cross Media Recommendations based on Linked Data Analysis." In: 2016.

[MD13] Fadi Maali and Stefan Decker. "Towards an RDF Analytics Language: Learning from Successful Experiences." In: *COLD*. 2013.

[MP15] Mico-Project. *MICO API Documentation*. Oct. 2015-2016. URL: http://mico-project.bitbucket.org/api/rest/ (visited on 10/30/2016).

[PD09] A. Phillips and M. Davis. *Tags for Identifying Languages*. RFC 5646 (Best Current Practice). Internet Engineering Task Force, Sept. 2009. URL: http://www.ietf.org/rfc/rfc5646.txt.

[PS04] Inc. Pivotal Software. *RabbitMQ - Messaging that just works*. Oct. 2004-2015. URL: https://www.rabbitmq.com/ (visited on 10/30/2015).

[RAR14] Josu Bermudez Rodrigo Agerri and German Rigau. "IXA pipeline: Efficient and Ready to Use Multilingual NLP tools." In: *Proceedings of the 9th Language Resources and Evaluation Conference (LREC2014) Reykjavik, Iceland* (2014).

[Red+16] J. Redmon et al. "You Only Look Once: Unified, Real-Time Object Detection." In: *IEEE Conference On Computer Vision And Pattern Recognition (CVPR)*. 2016.

[Sch+12] Sebastian Schaffert et al. "The Linked Media Framework: Integrating and Interlinking Enterprise Media Content and Data." In: *Proceedings of the 8th International Conference on Semantic Systems - I-SEMANTICS '12* (2012). URL: http://dl.acm.org/citation.cfm?id=2362504.

[Sch+14a] Kai Schlegel et al. "Balloon Synopsis: A jQuery Plugin to Easily Integrate the Semantic Web in a Website?" In: *Proceedings of the 2014 International Conference on Developers - Volume 1268*. ISWC-DEV'14. Riva del Garda, Italy: CEUR-WS.org, 2014, pp. 19–24. URL: http://dl.acm.org/citation.cfm?id=2878379.2878383.

[Sch+14b] Kai Schlegel et al. "Balloon Synopsis: A Modern Node-Centric RDF Viewer and Browser for the Web." In: *The Semantic Web: ESWC 2014 Satellite Events: ESWC 2014 Satellite Events, Anissaras, Crete, Greece, May 25-29, 2014, Revised Selected Papers*. Ed. by Valentina Presutti et al. Cham: Springer International Publishing, 2014, pp. 249–253. ISBN: 978-3-319-11955-7. DOI: 10.1007/978-3-319-11955-7_29. URL: http://dx.doi.org/10.1007/978-3-319-11955-7_29.

ISBN 978-3-902448-47-7

MICO Early Adopters

MICO unites leading research institutions from the information extraction, semantic web, and multi-media area with industry leaders in the media sector.

Salzburg Research
Coordinator, Austria

Fraunhofer
Germany

Insideout10
Italy

UMEA University
Sweden

University of Oxford
United Kingdom

University of Passau
Germany

Zaizi Ltd
United Kingdom

MICO is a research project partially funded by the European Union 7th Framework Programme (grant agreement no: 610480).

Images are taken from the Zooniverse crowdsourcing project Plankton Portal that will apply MICO technology to better analyse the multimedia content. https://www.zooniverse.org