Application of Semantic Web Technologies for (Semi-)Automatic Metadata Generation for Multimedia Data

Ruben Verborgh

Promotor: prof. dr. ir. Rik Van de Walle
Begeleiders: dr. Davy Van Deursen en Erik Mannens

Masterproef ingediend tot het behalen van de academische graad van
Master in de ingenieurswetenschappen: computerwetenschappen

Vakgroep Elektronica en informatiesystemen
Voorzitter: prof. dr. ir. Jan Van Campenhout
Faculteit Ingenieurswetenschappen
Academiejaar 2009–2010

UNIVERSITEIT
GENT

Word of Thanks

Firstly, I would like to thank Davy Van Deursen for devising the subject of this dissertation and for the highly professional way in which he supervised my work. He allowed me to give my creativity free rein while providing excellent support on crucial decisions. I would like to express my sincere appreciation to Professor Van de Walle for helping me outline my academic career, in which this dissertation is a milestone. Furthermore, I would like to thank Erik Mannens for initiating me into the Semantic Web.

My gratitude goes out to Jos De Roo for detailing the use of Semantic Web technologies at Agfa-Gevaert, his helpful e-mails and the excellent Euler software. His colleague Giovanni Mels showed me an operational application of SPARQL services. Thanks to Anna Hristoskova for the insights into OWL-S composition and for lending me her composer program. Joshua Tauberer did a magnificent job with the SemWeb library, which he allowed me to extend.

Last but not least, a heartfelt thank you to everyone that encouraged me during five intensive years of study, in particular my parents and Eddy Van Steenkiste. Special thanks to my mother for sustaining a pleasant study environment and to my father, who introduced me to computers as early as in 1993, when most things I write about were still in their infancy.

Dankwoord

In de eerste plaats wil ik Davy Van Deursen bedanken voor het aanbrengen van het onderwerp van deze masterproef en voor zijn professionele begeleiding. Hij gaf mijn creativiteit alle ruimte en stond met raad en daad klaar op cruciale momenten. Ik waardeer ten zeerste de steun van professor Van de Walle voor mijn academische carrière, waarin deze masterproef een belangrijke rol vervult. Daarnaast bedank ik ook graag Erik Mannens voor de inleiding tot het semantisch web.

Jos De Roo gaf mij een boeiende uiteenzetting over semantisch-webtechnologieën bij Agfa-Gevaert, hielp me verder via e-mail en is ook de auteur van de Euler-software. Hem en zijn collega Giovanni Mels, die me een actuele SPARQL-toepassing demonstreerde, ben ik veel dank verschuldigd. Eveneens aan Anna Hristoskova, voor haar werk rond OWL-S-composities, en Joshua Tauberer, voor de SemWeb-softwarebibliotheek, waaraan ik mee mocht ontwikkelen.

Tot slot bedank ik iedereen die er was voor me gedurende deze vijf jaar durende studie, in het bijzonder mijn ouders en Eddy Van Steenkiste. Ontzettende dank aan mijn moeder voor het gezellige studieklimaat thuis en aan mijn vader, die me al in 1993 met computers leerde werken, in een tijd waarin nog geen sprake was van de dingen waarover ik nu schrijf.

Usage Permission

Application of Semantic Web Technologies for (Semi-)Automatic Metadata Generation for Multimedia Data

by
Ruben Verborgh

Dissertation submitted for obtaining the degree of Master in Computer Science Engineering

Academic year 2009–2010

Ghent University
Faculty of Engineering
Department of Electronics and Information Systems
Chair: prof. dr. ir. Jan Van Campenhout

Supervisor: prof. dr. ir. Rik Van de Walle
Associate supervisors: dr. Davy Van Deursen and Erik Mannens

Summary

This dissertation investigates the application of Semantic Web technologies to the automatic metadata generation and annotation process for multimedia data. It describes the architecture of a generic semantic problem solving platform that uses independent algorithms to accomplish subtasks. A holistic vision on the metadata request harnesses more advanced problems, enabling the algorithms to interact with the solution as it is generated.

Samenvatting

Deze masterproef onderzoekt de toepassing van semantisch-webtechnologieën voor automatische metadatageneratie en -annotatie bij multimediale data. We beschrijven de architectuur van een algemene semantische probleemoplosser die onafhankelijke algoritmes gebruikt om deeltaken uit te voeren. Een holistische visie op metadataverzoeken maakt het aanpakken van complexere problemen mogelijk en laat de algoritmes interageren met de oplossing in wording.

Keywords: annotation, metadata generation, Semantic Web

Trefwoorden: annotatie, metadatageneratie, semantisch web

Application of Semantic Web Technologies for (Semi-)Automatic Metadata Generation for Multimedia Data

Ruben Verborgh
Supervisors: prof. dr. ir. Rik Van de Walle, dr. Davy Van Deursen, Erik Mannens

***Abstract*— This article discusses the application of Semantic Web technologies to the automatic metadata generation and annotation process for multimedia data. It describes the architecture of a generic semantic problem solving platform that uses independent algorithms to accomplish subtasks. A holistic vision on the metadata request harnesses more advanced problems, enabling the algorithms to interact with the solution as it is generated.**

***Keywords*—annotation, metadata, Semantic Web**

I. Introduction

Multimedia data requires an inherently more complex search process than textual data. To bridge this gap, content producers often provide *metadata* [1], the creation of which remains a time consuming activity. Several algorithms for automatic metadata extraction exist, but they are error-prone. Furthermore, only domain-specific algorithms have an intelligent vision on the object under annotation.

The *Semantic Web* [2] contains an enormous amount of knowledge in a great variety of domains, specified in a format that machines can interpret and process. In this paper, we will address the application of this knowledge to metadata generation.

Therefore, we create a general semantic problem solver, which answers a *request* about a specific *input*. Its architecture, based on the blackboard pattern [3] and depicted in Fig. 1, consists of several independent *collaborators* and a coordinating *supervisor*. These components communicate through a blackboard, which contains information in the *RDF* format because of the associated interoperability, extensibility and well-defined semantics.

II. Collaborators

Collaborators are entities that perform a specific task, which leads to the solution of the request or any of its subparts. Examples include feature extraction algorithms and semantic processors. Collaborators should be highly interoperable and able to communicate formally. Therefore, we choose RDF as their exchange format and implement them as SPARQL endpoints to allow distributed, transparent and flexible querying. SPARQL traditionally queries static information sources, yet its flexibility enables its use with on-demand information. The collaborator extracts the input from the query,

Ruben Verborgh is a Master student at Multimedia Lab, Department of Electronics and Information Systems, Ghent University, Belgium. E-mail: ruben.verborgh@ugent.be.

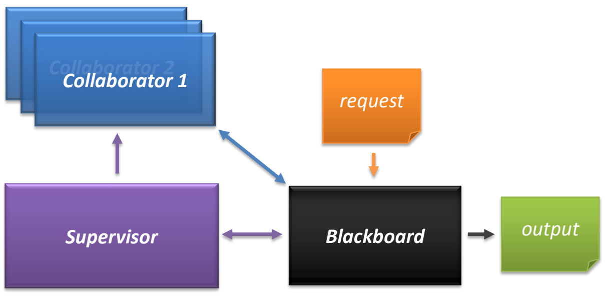


Fig. 1. Semantic problem solver architecture

generates output, and combines both into a virtual data source which is subsequently queried.

In order to decide whether a collaborator can aid in solving certain requests, we must dispose of a formal description of its capabilities and requirements. Since SPARQL endpoints are in fact web services, we selected the common specification *OWL-S* for these descriptions, to which we added support for SPARQL groundings. This format enables us to provide input and output parameters, as well as complex conditions on these parameters and various relations between them.

Existing algorithms can be transformed into collaborators by a wrapper, translating between RDF and their internal representation format. New algorithms could be implemented directly as SPARQL endpoints, eliminating the need for a proprietary interaction mechanism.

III. Composition

To solve complex requests, we need to combine several collaborators into an intelligent execution plan, containing several collaborator invocations. Since we use OWL-S to describe collaborators, this amounts to OWL-S service composition.

A first composition algorithm uses an *adjacency matrix* of the collaborators, built using an OWL-S matcher. Successive multiplication of the matrix with itself reveals which collaborators connect to each other. To construct a composition from an input to a request, we try to find services that match their conditions. Subsequently, we consult the adjacency matrix to determine whether a path between those services exists. In that case, we repeat the process iteratively to reconstruct the complete path. This algorithm has an excellent performance because many elements can be precomputed. Unfortunately, it can only generate so-called composition chains, in which every invocation step depends on exactly one other step. Furthermore, semantic knowledge can only be applied to every transition individually.

To overcome these limitations, we created an algorithm with a holistic vision on the problem, employing semantic knowledge on the problem in its totality. It performs the following steps:

1) convert the OWL-S description of the collaborators into *N3 rules* [4];
2) ask a *semantic reasoner* [5] to deduce the request from the input, triggering the collaborator rules;
3) construct compositions by verifying which rules the reasoner applied.

The reasoner has access to Semantic Web sources, providing ontological and rule-based knowledge. This process enables more sophisticated compositions that can serve more advanced requests.

IV. Supervision

The supervisor is the component responsible for composition selection, execution, progress display, failure handling and output formulation. When a request arrives, the supervisor selects a composition from the composer, based on several parameters such as available collaborators and required performance. It interprets this composition as a declarative program, whose execution order is solely governed by data dependencies. During execution, the supervisor maintains a variable binding which assigns blackboard values to variable names.

Two kinds of failure could occur with invocations: a collaborator error (caught by an exception mechanism) and incomplete output (when the collaborator does not return the expected information). We need to adapt the plan to obtain a successful result. The supervisor first determines the point of failure and its impact on the composition. It then tries to enrich the blackboard by deducing additional information using semantic data sources and inference mechanisms. Thereupon, it searches an additional composition, starting at the current state of the blackboard and ending in a resumption point. This composition is incorporated into the original plan.

Finally, the supervisor formulates an answer, based on the contents of the blackboard and the variable binding. Depending on the situation, we might be interested in intermediary results not explicitly requested.

V. Implementation

The above concepts have been implemented in a software platform dubbed *Arseco*. It offers facilities for the development of collaborators, either as a wrapper for existing algorithms or as a library for new implementations. The supervisor and composition algorithms were developed as a highly configurable framework, providing opportunities for research. All components were constructed in a modular way, to simplify the investigation of changes to specific parts. New knowledge sources and collaborators can be added instantaneously, accelerating the adaptation to other problem domains.

VI. Application

We considered the annotation of photographs as a use case. In one concrete example, the request demands to list names of people depicted in a certain photograph, which happens to contain several movie actors. We dispose of relevant collaborators and semantic knowledge.

The supervisor passes the input and request to the composer, which finds a sequence of a face detection and a face recognition algorithm. Then, the supervisor executes the first collaborator, resulting in the detection of four face regions. As a next step, it executes the face recognition collaborator, which only succeeds in recognizing two of the four faces. This incomplete output triggers failure handling.

The plan subsequently needs adaptation; first, the blackboard is enriched with information retrieved from DBpedia [6]. In this case, it yields statements about movies the actors appeared in, including cast and crew. From existing knowledge, the supervisor derives that the found colleagues of the recognized actors have a higher chance of appearing together on the photograph.

We resume with the execution of a second invocation of the face recognition collaborator, this time passing in a list of possible names. The collaborator uses this list to enhance its internal probabilities, and now manages to return the correct result. Finally, the supervisor formulates an answer including the names of the four actors.

This short example indicates how semantic knowledge can assist existing algorithms. Normally, the face recognizer would only be aware of the individual regions it analyzes. Thanks to the problem solving platform, it is able to interact with the solution as a whole, leading to a more complete end result.

VII. Future research

This paper demonstrates the potential of Semantic Web technologies in automatic metadata generation, illustrated by a practical application. For use in a real-world environment, several additional aspects should be considered. Topics for future research on the subject of semantic problem solving – and metadata generation in particular – include:

- definition of *quality metrics* to compare different metadata generation frameworks;
- handling *imperfect information*, since collaborator output is often the result of a heuristic process;
- investigation of advanced *knowledge modeling*;
- expansion of the concept to *different application domains*, such as video;
- examination of *user interaction*, as a source of feedback and for assisting in tasks which are currently performed better by humans.

Acknowledgements

The author would like to acknowledge the suggestions and feedback of Davy Van Deursen. Thanks to Jos De Roo for the introduction to services as SPARQL endpoints.

References

[1] J. R. Smith and P. Schirling, “Metadata standards roundup,” *IEEE MultiMedia*, vol. 13, pp. 84–88, 2006.

[2] T. Berners-Lee, J. Hendler, and O. Lassila, “The Semantic Web,” *Scientific American*, vol. 284, no. 5, p. 34, 2001.

[3] D. D. Corkill, “Blackboard systems,” *AI Expert*, vol. 6, no. 9, pp. 40–47, Sep. 1991.

[4] T. Berners-lee *et al.*, “N3Logic: A logical framework for the World Wide Web,” *Theory and Practice of Logic Programming*, vol. 8, no. 3, pp. 249–269, 2008.

[5] J. De Roo. Euler proof mechanism. [Online]. Available: http://eulersharp.sourceforge.net/

[6] Dbpedia. [Online]. Available: http://dbpedia.org/

Contents

List of Figures

List of Abbreviations

EXIF	Exchangeable Image File Format
HTML	HyperText Markup Language
HTN	Hierarchical Task Network
HTTP	HyperText Transfer Protocol
JPEG	Joint Photographic Experts Group
KIF	Knowledge Interchange Format
MIME	Multipurpose Internet Mail Extensions
MPEG	Moving Picture Experts Group
N3	Notation3
OWL	Web Ontology Language
OWL-S	OWL for Services
RDF	Resource Description Framework
RDFS	RDF Schema
SOAP	Simple Object Access Protocol
SPARQL	SPARQL Protocol and RDF Query Language
STRIPS	Stanford Research Institute Problem Solver
SW	Semantic Web
SWRL	Semantic Web Rule Language
URI	Uniform Resource Identifier
URL	Uniform Resource Locator
W3C	World Wide Web Consortium
WSDL	Web Services Description Language
XML	Extensible Markup Language

Chapter 1

Preface

1.1 Introduction

In the last decade, the world has witnessed an unprecedented growth of multimedia data production and consumption. On the one hand, the professional industry has widened its classic distribution network to include digital media platforms, adapting its business model to the on-demand needs of customers. On the other hand, an ever increasing number of amateur and semi-professional producers has found a path towards a massive audience. The philosophy of collaboration and information sharing – commonly referred to by the buzz term *Web 2.0* – has revolutionized the way we think about the Internet and media in general.

The recurring problem with the access to such tremendous amounts of data arises: how to instantaneously find the data you need? For textual data, this problem has been handled by many algorithms. Searching through other forms of multimedia data has traditionally been more difficult. A possible approach is to annotate this data with information that facilitates its retrieval, called metadata [39].

Metadata not only helps humans access the data they need, it also enables autonomous processing by machines for automated problem solving. This concept closely relates to the *Semantic Web* vision [5] – part of what is dubbed *Web 3.0* – as outlined by Tim Berners-Lee, in which machines become capable of analyzing increasingly more data on the Web. It consists of linked data in the RDF format [28], vocabularies or ontologies in the OWL format [32], query languages such as SPARQL [36], reasoning mechanisms such as N3Logic [6] and, on top of these, semantic applications.

These technologies unveil complex processing techniques when relevant metadata is available. Unfortunately, it is precisely this availability which forms the main obstacle. For professionals, metadata generation adds to the production cost because annotating requires tedious manual work. Amateur producers do not possess the necessary skills to provide metadata formally. Clearly, we need automated tools to assist with this cumbersome task.

1.2 Problem formulation

While automated feature extraction algorithms exist, they are prone to errors and lack an intelligent view on the object under annotation. Currently, people select the appropriate algorithms and parameters for a specific problem and initiate the process. As a result, manual intervention is required when a proposed solution does not meet the requirements. This approach lacks actual cooperation between different algorithms, which are unaware of their own abilities and limitations.

The question we will address in this dissertation is how human knowledge, represented in the Semantic Web, can aid the metadata annotation process of multimedia data. We would like to transcend the level of individual algorithms and evolve towards a more collaborative environment. Our answer should aim to narrow the current *semantic gap* between feature extraction algorithm output and human-centered annotation.

1.3 Goals

The first goal of this dissertation is to investigate how Semantic Web technologies can complement feature extraction algorithms to generate metadata. In this process, it is important that existing algorithms and knowledge can be reused to the extent possible. Adherence to relevant standards where applicable is essential.

The second goal is the construction of a metadata extraction framework which implements the developed techniques. This framework should demonstrate new and advanced abilities that are the result of the incorporation of Semantic Web knowledge.

Finally, as a third goal, we sketch paths for future research in the area of semantic multimedia annotation, covering the current limitations of the discussed techniques.

1.4 Methodology

To elaborate these goals, we develop a generic software framework for semantic problem solving, founded on an inherent theoretical basis. We realize the integration of different new and existing algorithms into the problem solving process. Furthermore, we examine the role of Semantic Web technologies in all aspects of the framework.

After that, we direct our focus towards metadata generation. In this dissertation, we only consider the annotation of images, yet strongly paying attention to the generalizability of our approach to other forms of multimedia content such as video. A concrete use case illustrates the potential of the concept.

1.5 Outline

Chapter 2 discusses the high-level architecture of the semantic problem solving framework. The adaptation of algorithms is considered in Chapter 3, and the way of combining them into a larger problem solving plan in Chapter 4. Chapter 5 handles the supervision of the solution process. Thereupon, Chapter 6 translates all these concepts into an actual software implementation.

While the preceding chapters treat generic problem solving, the use case in Chapter 7 is an application of metadata annotation. We explore future research possibilities in Chapter 8 and formulate a general conclusion in Chapter 9.

1.6 Conventions

We would like to familiarize the reader with several notations and standards used throughout this document.

1.6.1 RDF

Ontologies and vocabularies

RDF fragments are formatted in the Notation3 or N3 format [3] and typeset in a `fixed-width` font. Below is an example of an N3 fragment.

Example 1.1 Extract of an RDF music catalog in N3

```
@prefix mus: <http://example.org/MusicCatalog/>.
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#>.
@prefix xsd: <http://www.w3.org/2001/XMLSchema#>.

<Little_Red_Corvette> rdfs:label "Little Red Corvette";
                      a mus:Song;
                      mus:writtenBy <Prince>.
<Prince> rdfs:label "Prince";
         a mus:Singer,
           mus:Songwriter;
         mus:birthName "Prince Rogers Nelson";
         mus:birthDate "1958-06-07"^^xsd:date.
```

The above states that *Little Red Corvette* is a song by singer/songwriter *Prince*, born as *Prince Rogers Nelson* on June 7th, 1958. Note the use of prefixes to abbreviate namespaces of entities such as `<http://example.org/MusicCatalog/Song>`.

Rules

Notation 3 also defines the usage of N3Logic rules, which are a semantic representation of a logic implication with an antecedent and a consequent.

Example 1.2 Rule classifying obtuse triangles

```
@prefix geo: <http://example.org/Geometry/>.
@prefix math: <http://www.w3.org/2000/10/swap/math#>.
{
  ?polygon a geo:Polygon;
           geo:sideCount 3.
  ?corner geo:cornerOf ?polygon;
          geo:angle ?angle.
  ?angle math:greaterThan 90.
}
=>
{
  ?polygon a geo:ObtuseTriangle.
}.
```

This rule states that a three-sided polygon with an obtuse corner is an obtuse triangle.

Namespaces

In this document, we assume the presence of the following namespace prefix declarations in RDF fragments, and mostly omit them for brevity:

- `@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>.`
- `@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#>.`
- `@prefix owl: <http://www.w3.org/2002/07/owl#>.`
- `@prefix log: <http://www.w3.org/2000/10/swap/log#>.`
- `@prefix e: <http://eulersharp.sourceforge.net/2003/03swap/log-rules#>.`
- `@prefix ex: <http://www.example.org/>.` *(fictional namespace)*

We also omit declarations when their contents are obvious in a certain context.

1.6.2 OWL

OWL [32] is a format to describe vocabularies or ontologies of RDF resources. OWL documents are also expressed in RDF and can thus be formatted as N3. The standard describes relations between objects and predicates, such as inheritance.

1.6.3 SPARQL

SPARQL is both a query language [36] and a protocol [11] for querying RDF data sources. SPARQL queries are also typeset in a `fixed-width` font, and we assume the declaration of the aforementioned namespaces. The example below displays a typical `SELECT` query.

Example 1.3 Query for specific song names

```
PREFIX mus: <http://example.org/MusicCatalog/>.
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>.

SELECT ?songName
WHERE {
  ?artist mus:birthPlace "Minneapolis".
  ?song mus:writtenBy ?artist;
        rdfs:label ?songName.
}
```

This above query selects all songs names from artists that were born in Minneapolis. The contents of the `WHERE` clause are expressed in a subset of N3.

Another type of SPARQL queries create RDF graphs using the `CONSTRUCT` form.

Example 1.4 Query for a specific song and artist graph

```
PREFIX mus: <http://example.org/MusicCatalog/>.
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>.

CONSTRUCT { ?artist ?mus:authorOf ?song. }
WHERE {
  ?artist mus:birthPlace "Minneapolis".
  ?song mus:writtenBy ?artist.
}
```

This above query constructs all artist/song pairs from Minneapolis-born artists:

```
<Prince> ?mus:authorOf <Little_Red_Corvette>,
                       <Purple_Rain>.
<Jonathan_Edwards> ?mus:authorOf <Sunshine>.
```

The introduction of conventions finishes this introductory chapter. We now proceed with the platform architecture.

Chapter 2

Architecture

A first question arises undoubtedly in our mind: what exactly is metadata generation and how does it work? In this chapter, we examine the metadata generation process and the components involved. We devise an architecture for a semantic problem solving platform and investigate how existing technologies can play a role herein.

2.1 Metadata generation process

In order to design an architecture of a metadata generation framework, we must first have a firm grasp on the process inherent to metadata generation and its workings.

Definition 2.1 A ***metadata generation process*** is a process with two input parameters:

1. **item:** the multimedia item to be annotated and possibly pre-existing metadata;
2. **request:** a description of the demanded metadata;

and a single output parameter:

1. **metadata:** the generated metadata.

Pre-existing metadata may have been acquired upon the creation of the multimedia item [43] or can be the result of earlier metadata generation efforts.
We do not know in advance whether all requested metadata will be found. In addition, the result may contain unrequested but valuable metadata as a side effect.

Example 2.1 Photograph description

A photograph of a meeting must be accompanied by a textual description. We dispose of a JPEG image which contains the image contents (*item*) and EXIF information (*pre-existing metadata*). We start the metadata generation process with a request for a textual description. The process finds a textual description, as well as the names of the people in the photograph (*metadata*).

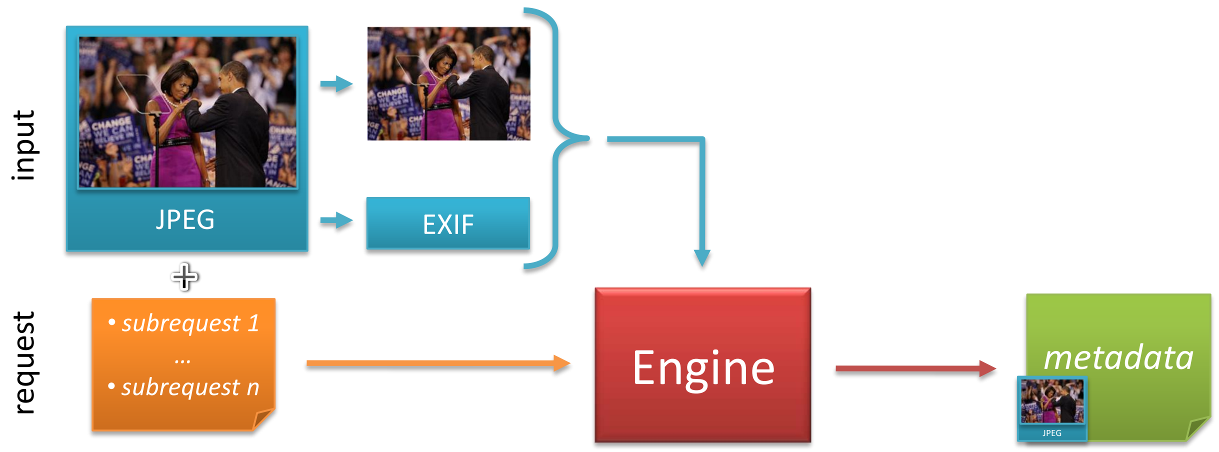


Figure 2.1: High-level overview of the metadata generation process and its execution

2.2 Process engine

2.2.1 Structure

A black box view of the execution of the photograph description example is depicted in Figure 2.1. As shown, the metadata generation process is governed by a process engine.

> **Definition 2.2** A ***metadata generation process engine*** is a platform that executes a metadata generation process.

The required versatility of a process engine makes a monolithic structure impossible. To manage its complexity, we separate the coordination task from the actual processing and define a supervisor component.

> **Definition 2.3** A ***supervisor*** is the principal component of a metadata generation process engine. Its tasks consist of:
>
> 1. *decomposition* of the request into subtasks;
> 2. *delegation* of these subtasks;
> 3. *coordination* of these subtasks.

The supervisor strives to return metadata that fulfills the request and its subrequests as completely as possible. The subtasks have to be executed by separate, highly specialized entities called collaborators.

> **Definition 2.4** A ***metadata collaborator*** is a component that, given an input, performs an information-generating processing task, that is likely to head towards the solution of an explicit or implicit part of a metadata request.

2.2.2 Internal metadata representation

It is our intention to incorporate semantic knowledge into the metadata generation process. Consequently, we need to select an internal representation format for metadata which enables this functionality. We have chosen RDF [28] for the following reasons:

- **interoperability:** RDF is a simple, triple-based format with different representations;
- **modularity:** information can be divided across multiple parts;
- **extensibility:** new types of information, regardless of complexity, can be modeled;
- **well-defined semantics:** in contrast to XML, RDF has rigorous semantics [2];
- **reasoning:** several RDF reasoners exist, which enable deduction of new knowledge;
- **existing knowledge:** the Semantic Web contains a vast amount of semantic knowledge in RDF format which can be applied during the process.

As an additional advantage, the multimedia content description standard MPEG-7 [31] can be serialized as RDF [19, 27, 44]. Existing MPEG-7 metadata can directly serve as input for the engine and MPEG-7 output can be stored as is. In case the item and request parameters have another format, they must be adapted to RDF such as in the example below.

Example 2.2 Photograph description input parameters

Original input parameters

- **item:** an image with URL *http://www.example.org/photo.jpg*
- **request:** a textual description of the image

RDF input parameters

- **item:** `<http://www.example.org/photo.jpg> a <http://xmlns.com/foaf/0.1/Image>.`
- **request:**
```
<http://www.example.org/photo.jpg>
        <http://purl.org/dc/elements/1.1/description> ?description.
```

The example also introduces us to the *variable* notation for requested entities. Here, we indicate that a request is fulfilled if a binding for the `description` variable has been found. Of course, this offers no guarantee that the actual result includes this information.

A interesting feature of the RDF format is that there are many ways of representing the same information, due to the existence of numerous ontologies. On the downside, the difficulty of combining different information sources arises. The ontologies used to describe the metadata should be well-defined and connected to each other. OWL [32] contains mechanisms to describe various complex relations between different ontologies. Considerable and ongoing efforts to improve these relations have been made by the *Linking Open Data Community Project* [7].

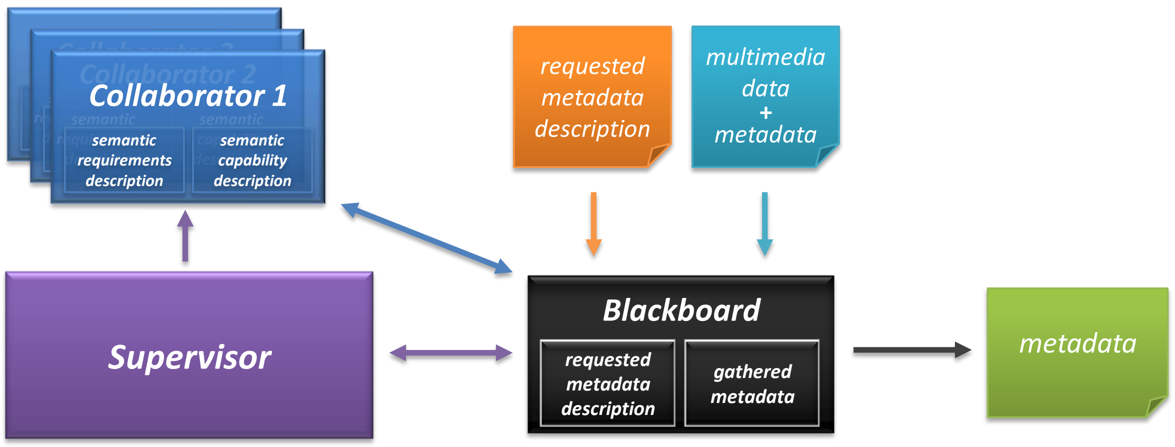

Collaborator 2
Collaborator 1
semantic requirements description
semantic capability description
requested metadata description
multimedia data + metadata
Supervisor
Blackboard
requested metadata description
gathered metadata
metadata

Figure 2.2: Internal structure of the process engine, revealing a blackboard pattern

2.2.3 Blackboard architectural pattern

The framework being designed thus consists of a supervisor and several independent collaborators who communicate by means of an internal data representation. This concept maps directly onto the *blackboard* architectural pattern widely used in artificial intelligence applications [12]. Applied to our framework, it leads to the engine structure of Figure 2.2.

The metadata generation process is executed using the following iterative algorithm.

Algorithm 2.1 Metadata generation process execution

```
Require: $input, request, collaborators$
  $blackboard \leftarrow input$
  $collaborators_{unused} \leftarrow collaborators$
  repeat
    $collaborators_{matching} \leftarrow$ find collaborators in $collaborators_{unused}$
                                  that, given $blackboard$, lead to $request$
    if $collaborators_{matching} \neq \emptyset$ then
      $collaborator \leftarrow$ choose collaborator from $collaborators_{matching}$
      $result \leftarrow collaborator.execute(blackboard)$
      $blackboard \leftarrow blackboard \cup result$
      $collaborators_{unused} \leftarrow collaborators_{unused} - \{collaborator\}$
    end if
  until ($blackboard$ satisfies $request$) $\vee$ ($collaborators_{matching} = \emptyset$)
  return $blackboard$
```

In essence, we repeatedly try to invoke collaborators until none of them has any new information to contribute. This algorithm only represents a highly abstracted view of the actual algorithm executed by the supervisor, as outlined in Chapter 5. In reality, the collaborator invocations will proceed in an orderly fashion. Also, most subproblems – such as the search for a collaborator and its selection – are non-trivial.

2.3 Components

2.3.1 Blackboard

By analogy with a real-world blackboard, the blackboard component is a common space where the supervisor and different collaborators interact with each other. It keeps track of the metadata that needs to be found and the metadata that has already been gathered. Initially, the former is bound to the request parameter and the latter to the input parameter. All content is stored and retrieved as RDF statements.

In a traditional system, the supervisor would have *read-only* access to the blackboard to decide which collaborator to invoke. Collaborators in turn would have *write-only* access for adding output, given their input is passed by the supervisor. The presence of additional arrowheads in Figure 2.2 however, indicates two significant changes.

1. The supervisor also has *write* access to **enrich the blackboard**'s RDF content using semantic reasoning and existing semantic knowledge.

2. Collaborators also have *read* access to **reuse information** gathered by previous collaborators, which they can employ to enhance their own functionality. This important concept is called ***semantic feedback***.

> **Example 2.3** Usage of semantic feedback for traffic sign recognition
>
> Given a JPEG photograph of a traffic situation, a collaborator could extract the geographic coordinates from the EXIF data and map them to a country, for instance Australia. Next, a traffic sign collaborator could try to recognize signs. Thanks to the country information, the collaborator can limit its search to Australian traffic signs, which greatly improves the recognition process in terms of performance and accuracy.

It should be noted that the supervisor decides what blackboard information each collaborator can access. This is necessary for *efficiency* (irrelevant data must not consume any bandwidth), *focus* (the collaborator needs to operate on specific data) and *confidentiality* (the collaborator may only access data needed for correct operation). Also, this simplifies the implementation of collaborators since they do not need to actively request data.

The blackboard itself can be implemented in an elementary way as an RDF triple store. Depending on the problem size and required performance, this store may reside fully or partially in memory or on disk. Many implementations of RDF stores already exist [42], so I will refrain from discussing this any further.

2.3.2 Collaborator

Definition 2.4 of a collaborator covers a rather broad spectrum of components. Some essential collaborator classes are listed non-exhaustively below.

- **Feature extraction algorithms:** the classic range of signal processing algorithms capturing various characteristics of multimedia content.
  Examples: *image segmentation, speech recognition, scene detection*
- **Semantic algorithms:** algorithms that operate solely on the semantic content of the blackboard, trying to infer new knowledge. This process can be driven by ontologies or semantic rules, both online and offline.
  Examples: *content relationship search, textual description generation*
- **Human collaborators:** tasks that are traditionally solved better by humans.
  Examples: *foreground separation, trajectory matching, facial expression recognition*

As a consequence, the implementation of a collaborator should be completely independent of low-level concerns such as programming language or operating system. This can be realized by abstractions and adapters. Since we cannot guarantee that we will interact with an actual program, those abstractions must also account for human-machine interaction. The operational aspects of collaborators are tackled in Section 3.1.

It is apparent that variations between different aspects of collaborators (such as capability, performance, cost, reliability, availability...) can be substantial. The supervisor should take this into account during its selection step, choosing the collaborator that best satisfies the problem constraints. This requires a formal description of the collaborator's capabilities and requirements, as described in Section 3.2.

2.3.3 Supervisor

The complexity and heterogeneity of the associated collaborators suggest that the supervisor must be a sophisticated component. The decomposition, delegation, and coordination tasks listed in Definition 2.3 are translated into

- the creation of an **execution plan** (Chapter 4);
- the actual **execution** of this plan (Section 5.3);
- adequate **response** to various outputs and errors (Section 5.5).

A high-quality supervisor must be reusable across substantially different situations. The types of tasks it can handle should only be limited by the collaborators and knowledge it has access to. Therefore, it is again important to implement this component on a level which abstracts those details. Supervision is considered in Chapter 5.

Chapter 3

Collaborators

Chapter 2 gave a definition of a collaborator and sketched the environment in which it is to be used. In this chapter, we will focus on the interaction with their environment and the description of their capabilities. Concrete examples will give an impression of the extensive possibilities collaborators can offer.

3.1 Interaction

3.1.1 Representation format

The tasks performed by collaborators span a very wide range, so finding a comprehensive interaction model poses a challenge. Input and output parameters must be precisely specified, at the same time leaving room for new, previously unused parameters. A list of required properties for collaborator interaction follows:

- **interoperable:** enabling communication with various other components, regardless of low-level specifics such as operating system and programming language;
- **flexible:** handling a variety of inputs and outputs;
- **formal:** able to communicate in a formal way with well-defined semantics.

The concept should be portable to other contexts instead of solely serving the goals of our problem solving platform. Of course, we must ensure that integration is straightforward. Looking at the above requirements and comparing them to the features discussed in Subsection 2.2.2, we conclude that RDF fits this purpose well.

The usage of a formal representation format counters the issue in many of today's processing algorithms – possible collaborators in our platform – of having a proprietary input and format, whether XML-based or not. Integrated semantics facilitate the automated interpretation and exchangeability of results.

3.1.2 Communication protocol

Having chosen RDF as representation format, we still need to decide how communication with a collaborator takes place. This communication protocol should meet the following design requirements:

- **distributed:** access to collaborators located on different machines;
- **flexible:** ability to specify variations on input and outputs;
- **transparent:** identical behavior, regardless of varying properties such as physical location and technological differences;

These requirements obviously hint at a service-oriented architecture [35]. Therefore, we consider collaborators as web services. A classic web service communication protocol choice is SOAP [22]. However, this protocol is rather verbose and does not account for sufficient flexibility: input and output parameters are passed in a rigid structure that does not allow variations.

Given the use of RDF and the definition of collaborators as information-generating entities, we can conveniently implement them as SPARQL endpoints. SPARQL is a query language and protocol used to retrieve information from semantic data sources, traditionally static databases of RDF content. Examples include *DBpedia* [8], *BBC Programmes and Music* [17], and the *Linked Movie Data Base* [23]. A collaborator is essentially a data source, which does not necessarily offer predefined information, but creates new information in a demand-driven way. We now look at querying techniques for such on-demand data sources.

Classic SPARQL queries for static data sources

The concept behind SPARQL is similar to that of other query languages: to retrieve a specific view on a larger data collection. The query expresses our constraints.

Example 3.1 Finding the name of a person born on October 22th, 1811

```
PREFIX dbpedia-owl: <http://dbpedia.org/ontology/>
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>
CONSTRUCT { ?person dbpedia-owl:birthName ?name. }
WHERE {
  ?person a dbpedia-owl:Person;
          dbpedia-owl:birthDate "1811-10-22"^^xsd:date;
          dbpedia-owl:birthName ?name.
}
```

Note how this query merely conditions its own output, instead of being passed the input *"a person born on 1811/10/22"*. This aligns with the way we think about the underlying RDF database: the `WHERE` clause is a *filter* that retains all triples matching the specified template.

Classic SPARQL queries for collaborators

A first approach would be to query collaborators in the exact same way. Suppose we have access to a collaborator that implements the following informal description.

> **Example 3.2** Color mixing collaborator
>
> - **Input:** a number of colors with a specified name
> - **Output:** a color with a specified name that is a mix of the input colors

The first step is to find formal types for the parameters. In practice, we would search for existing color ontologies. For now, we will simply create our own.

> **Example 3.3** Color mixing collaborator with RDF types
>
> - **Input:** a number of `Color` entities with a `hasColorName` property
> - **Output:** a `Color` that has a `hasColorName` property, and a `isMixOf` property with the list of input colors as value

If we want to ask the result of mixing red and yellow, we could invoke the collaborator using the following query.

> **Example 3.4** Color mixer invocation with classic SPARQL

```
CONSTRUCT { ?color :hasColorName ?colorName. }
WHERE {
  ?colorA a :Color;
          :hasColorName "red".
  ?colorB a :Color;
          :hasColorName "yellow".
  ?color a :Color;
         :hasColorName ?colorName;
         :isMixOf (?colorA ?colorB).
}
```

As opposed to querying data source, invoking a collaborator is not as such about filtering, but about retrieving the outputs of a process on the inputs. Although it would be possible to build a collaborator this way, there are several drawbacks to this approach:

- **missing indication** that the query is a collaborator invocation;
- **unclear distinction between input and output** in the `WHERE` clause;
- **conceptual mismatch** by using output conditions as opposed to input parameters.

Clearly, we need a more advanced query prototype which makes the invocation explicit.

SPARQL queries with explicit invocation

We consider the collaborator as a virtual data source and refer to each invocation by a dedicated `Request` entity. The input and output parameters of the invocation are parameters of this entity. The inputs are generally threated as known values; the outputs are usually unbound variables. Inside the SPARQL query, we can refer to this entity and its associated properties. This enables us to invoke a collaborator while keeping all the benefits of a SPARQL query and its declarativeness.

The SPARQL query in the next example shows how this technique overcomes the previous weaknesses, clearly indicating the invocation, its parameters, and their direction.

Example 3.5 Collaborator invocation with a `Request` entity

```
PREFIX sr: <http://example.org/SparqlRequest.owl#>
CONSTRUCT { :mixedColor :hasColorName ?colorName. }
WHERE {
  [a sr:Request;
   sr:method "MixColor";
   sr:input [a :Color;
             :hasColorName "red"],
            [a :Color;
             :hasColorName "yellow"];
   sr:output [a :Color;
              :hasColorName ?colorName]]
}
```

The collaborator maintains an entity corresponding to the `Request` in the `WHERE` clause, containing the same information as the entity in the query. The collaborator executes its task on the entity `input` values; its computed output gets bound to the entity `output` values. The resulting `Request` entity is subsequently queried in its entirety.

We highlight the fact that this `Request` entity is purely virtual: it is only accessible during the execution of the query and remains invisible to other clients accessing the collaborator at the same instant. The following query, which always returns zero results, illustrates this.

Example 3.6 Querying a non-existent `Request` entity

```
PREFIX sr: <http://example.org/SparqlRequest.owl#>
SELECT ?input
WHERE {
  [a sr:Request;
   sr:input ?input]
}
```

Result: *0 items*

SPARQL queries with explicit invocation and named parameters

The previous example has two symmetric input parameters. To generalize the query prototype to a broader class of collaborators, it is necessary to provide parameter identification for input and output. This is done by setting the value of the `input` and `output` to a `ParameterValue` entity, instead of solely the actual value. An example is integer division.

Example 3.7 Collaborator invocation with named parameters

```
PREFIX sr: <http://example.org/SparqlRequest.owl#>
CONSTRUCT { :answer :hasValue ?quotient. }
WHERE {
  [a sr:Request;
   sr:method "Divide";
   sr:input [a sr:ParameterValue;
             sr:bindsParameter "dividend";
             sr:boundTo 15],
            [a sr:ParameterValue;
             sr:bindsParameter "divisor";
             sr:boundTo 5];
   sr:output [a sr:ParameterValue;
              sr:bindsParameter "quotient";
              sr:boundTo ?quotient]]
}
```

For simplicity, the query can be abbreviated by removing parts of the `WHERE` clause that are known in advance. A request will always have type `Request`, and a parameter will always be of type `ParameterValue`, so these statements can be omitted. The method name is also unnecessary, because a collaborator only has a single task and is already identified the moment it receives the query. However, if a query is viewed out of its usual context, these items clarify its intended meaning.

Collaborator invocations where the distinction between arguments is clear (e.g. different types) or unimportant (e.g. symmetric) can choose to support both query prototypes in a straightforward way. Named parameters can be converted into unnamed parameters using a semantic rule, removing the extra level of indirection introduced by `ParameterValue`.

Example 3.8 N3 rule converting named parameters into unnamed parameters

```
@prefix sr: <http://example.org/SparqlRequest.owl#>.
{ ?request ?hasParam [a sr:ParameterValue;
                      sr:boundTo ?paramValue]. }
=>
{ ?request ?hasParam ?paramValue. }.
```

SPARQL queries with complex parameters

In case the input or output parameters are complex, it is possible to specify either of them as *N3 strings*, which are string representations of RDF graphs serialized as N3. They may be required when the structure of the output RDF graph is unknown in advance, making it impossible to reserve sufficient variables for retrieval. Furthermore, complex graphs – such as those containing reified statements – are also better represented by N3 strings. The Notation3 specification has been extended with this functionality in the `log` namespace[1].

The next example, finding all descendants of a person in a genealogical tree, illustrates complex output. We do not know beforehand how deeply the result graph will be nested and SPARQL does not provide facilities to capture *all* descending nodes. Therefore, we need an N3 string to store the result, which must be parsed afterwards to obtain the corresponding RDF graph. For the sake of example, we also supply the input parameter as an N3 string, although not absolutely necessary.

Example 3.9 Collaborator invocation with complex parameters

```
PREFIX sr: <http://example.org/SparqlRequest.owl#>
PREFIX log: <http://www.w3.org/2000/10/swap/log#>
SELECT ?descendantsN3 WHERE {
  [a sr:Request;
   sr:method "FindDescendants";
   sr:input [a log:Formula;
              log:n3String "<Franz_Liszt> a :Person."];
   sr:output [a log:Formula;
               log:n3String ?descendantsN3]]
}
```

N3 strings also offer collaborators the freedom to return more expressive values. Consider an algorithm that recognizes words in an audio fragment. In the strict case, the return value is ill-defined when uncertainty between two alternatives arises. RDF enables us to express this uncertainty semantically, yet this will only fit in an N3 string output variable.

Example 3.10 Expressing uncertainty in collaborator output

```
@prefix e: <http://eulersharp.sourceforge.net/2003/03swap/log-rules#>.
({<speech.wav#t=5.38,5.71> :containsWord "summer".}
 {<speech.wav#t=5.38,5.71> :containsWord "sombre".}) e:disjunction [a e:T].
```

This means the disjunction of “the fragment contains *summer*” and “the fragment contains *sombre*” holds, which by elementary logic implies at least one of these must hold.

[1]The `log` namespace is located at `http://www.w3.org/2000/10/swap/log#`.

3.1.3 Algorithm conversion

Having defined a formal model for invocations, we now direct our attention to a conversion method from algorithm to collaborator. When designing a program for an algorithm, an author arrives at a point where the output format must be decided. Often, a proprietary format is created, accompanied by an informal description. In this case, it would be convenient to choose RDF since this gives access to an existing formal model, compliant with other information in the Semantic Web. Should the author proceed with another format – proprietary or standardized – then an adapter needs to be written to convert the input and output into RDF.

The program now communicates entirely using RDF. A SPARQL query engine, wrapped around the program, processes the virtual `Request` entity in a way similar to the following, turning the collaborator into a complete SPARQL endpoint.

Algorithm 3.1 Collaborator SPARQL wrapper

```
Require: $query, collaborator$
  $inputRdf \leftarrow query.whereClause$
  $outputRdf \leftarrow collaborator.process(inputRdf)$
  $requestRdf \leftarrow joinRdfGraphs(inputRdf, [$rs:output$, outputRdf])$
  $result \leftarrow sparqlEngine.executeQueryOnGraph(query, requestRdf)$
  return $result$
```

This process is visualized in the image below. The RDF inputs are extracted from the query (1) and passed to the collaborator (2), which generates RDF output (3). Input and output form the completed `Request` entity (4). Thereupon, the query is executed on this entity (5), yielding the final result.

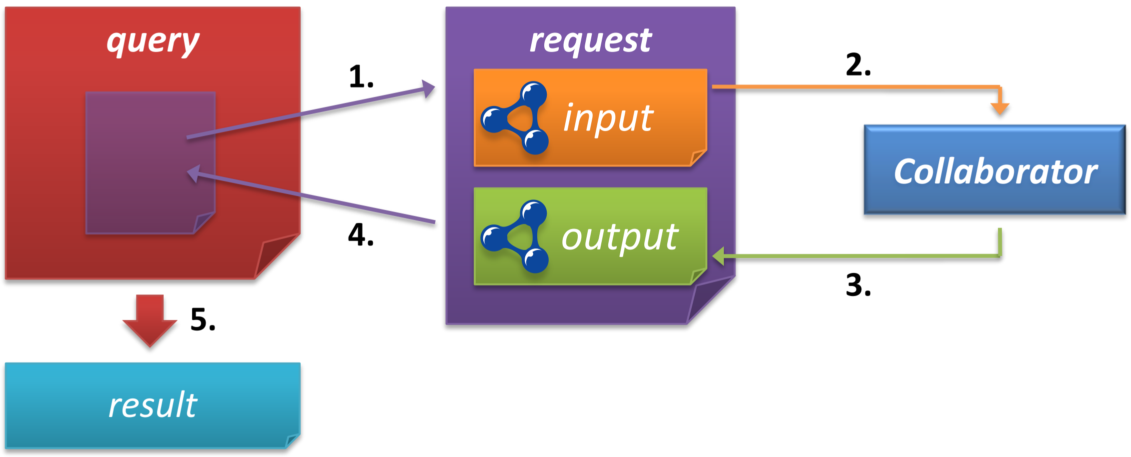


Figure 3.1: Collaborator SPARQL wrapper process

3.2 Capabilities and requirements description

3.2.1 Description method

An automated problem solving process needs to be able to decide which collaborators it needs. The classes of problems it handles, as well as the collaborators used for this task, may vary over time. This functionality requires an efficient and formal collaborator description mechanism that is compatible with the problem description. When we realize that SPARQL endpoints are in fact web services, we can describe them as such. A common RDF-compliant specification for semantic web service descriptions is *OWL-S* [30], which consists of a three-part paradigm:

- **profile:** a brief description of the service's capabilities and requirements;
- **model:** service usage modalities and a more in-depth description of its capabilities and requirements, suitable for service composition;
- **grounding:** technical details on communication with the service.

The relevant parts for collaborator description are *model* (to describe the task of the collaborator) and *grounding* (to describe its SPARQL endpoint details). The *profile* part is not as important, yet OWL-S requires its presence; fortunately it is constructed easily from the model parameters. In the next subsections, I will elucidate the different parts through an example collaborator that recognizes a face in an image region.

Example 3.11 Informal collaborator description

- **Input:** a region which depicts a face
- **Output:** the recognized face and the depicted person

This informal description leaves room for interpretation and should be formalized. If the collaborator author already formalized the input and output parameter model in RDF, we can copy this into the service description. However, it will often be more convenient to create the formal service description first, deciding which RDF classes to use for input and output, and then use this description to implement the actual RDF parameters of the collaborator. This corresponds to the *design by contract* methodology in software engineering.

3.2.2 Profile

As the profile is of little interest for our application, we simply declare some basic information – useful for human interpretation – and reuse the input and output parameters that our model will define later on. This approach is also suggested in [30, section 4.1], highlighting the fact that despite their different role, profile and process represent the exact same service.

The following profile is sufficient for the face recognition example.

Example 3.12 Collaborator profile description

```
@prefix Profile: <http://www.daml.org/services/owl-s/1.1/Profile.owl#>.
:FaceRecognitionProfile a Profile:Profile;
                        Profile:serviceName "Face Recognition Collaborator";
                        Profile:textDescription "Collaborator that recognizes
                                                 a face in region.";
                        Profile:hasInput :RegionInput;
                        Profile:hasOutput :FaceOutput;
                        Profile:hasOutput :PersonOutput;
                        Profile:has_process :FaceRecognitionProcess.
```

3.2.3 Process

The process needs to contain detailed information about all parameters. I will start with a basic description and extend it as inadequacies come to light.

Example 3.13 Basic collaborator process description

```
@prefix Process: <http://www.daml.org/services/owl-s/1.1/Process.owl#>.
:FaceRecognitionProcess a Process:AtomicProcess;
                        Process:hasInput :RegionInput;
                        Process:hasOutput :FaceOutput;
                        Process:hasOutput :PersonOutput.

:RegionInput a Process:Input;
             Process:parameterType "http://example.org/Images.owl#Region".
:FaceOutput a Process:Output;
            Process:parameterType "http://example.org/Faces.owl#Face".
:PersonOutput a Process:Output;
              Process:parameterType "http://example.org/Persons.owl#Person".
```

Although this profile seems correct on the surface, it does not convey all intended semantics for a reliable description of a face recognition activity. Consider a collaborator that, regardless of the input it receives, always returns the exact same predefined person in the `PersonOutput` parameter. This collaborator does not recognize faces, yet it fully complies with the description of the example above. Also, there is no guarantee whatsoever that the region in `RegionInput` actually contains a face; it could depict anything or nothing at all. This means that even an actual face detection algorithm could fail to return a correct result.

To obtain a process description that fits the collaborator, we need to correct the following occurring problems:

- the input is not guaranteed to contain a face
  ⇒ *the **input constraints** must be specified rigorously;*
- the face in the input is not necessarily that of the person in the output
  ⇒ *we require a **semantic relation** between different input and output parameters.*

We can capture these additional semantics by using preconditions and postconditions.

Preconditions

OWL-S supports the use of preconditions to enforce input constraints that go beyond RDF types. These conditions are traditionally expressed in languages such as KIF [20] or SWRL [24]. We have opted to use N3 rules [6], which are very powerful due to the possibility of more sophisticated built-in functions [4] and the existence of advanced reasoners [14].

Therefore, we needed to extend the `Expression` ontology to include support for Notation3 expressions and conditions. The extended ontology `owl-s-sparql`, which makes it possible to express restrictions in plain RDF literals, can be found in Appendix A. The description example is supplemented with the following preconditions.

Example 3.14 Collaborator process description preconditions

```
@prefix Process: <http://www.daml.org/services/owl-s/1.1/Process.owl#>.
@prefix owl-s-sparql: <http://www.example.org/owl-s-sparql.owl#>.
@prefix Expr: <http://www.daml.org/services/owl-s/1.1/generic/Expression.owl#>.
:FaceRecognitionProcess Process:hasPrecondition :RegionPrecondition.

:RegionPrecondition a owl-s-sparql:N3-Expression;
                    Expr:expressionBody
  "?regionInput <http://example.org/Images.owl#regionDepicts> ?face.
   ?face a <http://example.org/Faces.owl#Face>.".
```

Input and output parameters are referred to by *parameter variables* whose name is the last segment of the parameter URI, lowercasing the first letter. Thus `?regionInput` refers to the parameter named `http://example.org/FaceDetectionCollaborator.owl#RegionInput`. In case of ambiguity, the description document can provide parameter name aliases. This technique is similar to that of SWRL variables in OWL-S [30]. However, a rigorous mechanism that links parameters to variable names should be created. This is left as future work.

Also note the introduction of custom variables (`face`) in the precondition. These are ordinary variables, with the additional benefit that they remain bound in the postconditions, where they can be used in addition to the input and output variables.

Postconditions

In a similar fashion, relations between input and output parameters can be expressed in the Notation3 format. OWL-S terminology defines postconditions as effects of a service result. It is possible to specify multiple results that account for different cases, such as normal and erroneous execution. For simplicity, we will limit the face recognition example to a single result and effect.

Example 3.15 Collaborator process description postconditions

```
@prefix Process: <http://www.daml.org/services/owl-s/1.1/Process.owl#>.
@prefix owl-s-sparql: <http://www.example.org/owl-s-sparql.owl#>.
@prefix Expr: <http://www.daml.org/services/owl-s/1.1/generic/Expression.owl#>.
:FaceRecognitionProcess Process:hasResult :FaceRecognitionResult.

:FaceRecognitionResult a Process:Result;
                        Process:hasEffect :FaceRecognitionEffect.

:FaceRecognitionEffect a owl-s-sparql:N3-Expression;
                        Expr:expressionBody
  "?regionInput <http://example.org/Images.owl#regionDepicts> ?faceOutput.
   ?faceOutput <http://example.org/Faces.owl#isFaceOf> ?personOutput.".
```

The last two examples indicate that a few lines of RDF can be sufficient to model the complex input/output relations and restrictions necessary to represent collaborators adequately.

3.2.4 Grounding

The remaining part of the collaborator description details its SPARQL endpoint properties. To access a SPARQL endpoint, we need at least the following details:

- the **URL** of the endpoint;
- the **SPARQL versions** supported;
- the **SPARQL query forms** supported (for collaborators usually `CONSTRUCT` and `SELECT`).

Unfortunately, OWL-S only provides built-in support for WSDL service groundings. At the moment of writing, a draft by the W3C SPARQL Working Group on endpoint descriptions exists, but it is not linked to the OWL-S ontologies [47, 48]. An interesting approach is to create a service grounding, compatible with OWL-S, that is linked to the SPARQL endpoint description. The `owl-s-sparql` ontology listed in Appendix A offers this functionality by uniting the definitions of service grounding and SPARQL endpoint. Common grounding properties are provided by OWL-S. SPARQL endpoint specific properties are imported from the service description ontology of the W3C draft. Properties that are currently missing, such as supported SPARQL versions and query forms, are described in `owl-s-sparql`.

The example below displays a possible grounding for the face detection collaborator. It can coexist side by side with supplementary groundings, in case the collaborator supports other protocols such as WSDL.

Example 3.16 Collaborator SPARQL grounding description

```
@prefix sd: <http://www.w3.org/2009/sparql/service-description#>.
@prefix owl-s-sparql: <http://example.org/owl-s-sparql.owl#>.
:FaceRecognitionGrounding a owl-s-sparql:SparqlServiceGrounding;
          owl-s-sparql:supportsQueryForm owl-s-sparql:SparqlQueryFormConstruct,
                                         owl-s-sparql:SparqlQueryFormSelect;
          owl-s-sparql:supportsSparqlVersion owl-s-sparql:SparqlVersionQuery1_0;
          sd:url <http://example.org/collaborators/FaceRecognition/sparql>.
```

The above information makes it possible to invoke SPARQL endpoint operations on the collaborator through the specified URL, as specified by the SPARQL protocol [11].

3.2.5 Service

Finally, the three service parts need to be stitched together in an OWL-S service construct. We conclude this chapter with the remainder of the example collaborator description.

Example 3.17 Collaborator service description

```
@prefix Service: <http://www.daml.org/services/owl-s/1.1/Service.owl#>.
:FaceRecognitionService a Service:Service;
                        Service:describedBy :FaceRecognitionProcess;
                        Service:presents :FaceRecognitionProfile;
                        Service:supports :FaceRecognitionGrounding.
```

Having specified the operational aspects of collaborators and a means of formally describing their functionality, we are able to integrate them in complex problem solving systems. The next chapter instructs how the collaborator descriptions can be used to generate compositions that handle advanced requests.

Chapter 4

Composition

The strengths of individual collaborators do not meet the demands of complex problems. Instead, they should be combined in an intelligent way to achieve more sophisticated goals. This chapter discusses methods to create these powerful collaborator compositions.

4.1 Planning and scheduling

4.1.1 State of the art

Every entity that uses a *plan* to solve problems it has never encountered before, exhibits a certain degree of intelligence. This activity, which distinguishes humans from most other life forms, has been an important research topic in the artificial intelligence domain since its existence. *Automated planning* is the process of scheduling subtasks to reach a specified end state from a certain start state. Several planning algorithms, such as STRIPS [16] and HTN [15], are still active research topics.

For a successful planning process, the problem and available actions should be described extensively. Since the actions at our disposal are those performed by collaborators, we have access to their OWL-S descriptions. The OWL-S standard was specifically designed to allow composition and interaction of services. Not surprisingly, several tools for service composition have been created [25]. On the one hand, *service matchers* answer the question whether two services can succeed each other. On the other hand, *service composers* generate a sequence of services to accomplish a specified goal.

Unfortunately, interest in this matter has stagnated and most tools are outdated because of their incompatibility with current OWL-S versions. Ongoing research at the Ghent University [26] permitted me nevertheless to test the composability of collaborators against OWL-S standards. Although these composers could construct several collaborator compositions, it was clear that more sophisticated compositions required more advanced techniques, which is the topic of the present chapter. Usage of web service composition to solve multimedia problems has already been suggested in [21, 40, 41].

4.1.2 Approaches

There are two main approaches to service composition with particular advantages and disadvantages, as listed below.

- **Independent composition algorithm and specific matcher.** The algorithm is unaware of any internal details of the services it composes. A separate detail-aware matcher indicates whether two services can be scheduled in succession. Advantages are simplicity, performance, and reusability, at the cost of limited problem solving capabilities.

  Example: a service matcher for OWL-S descriptions is plugged into a combinatorial algorithm, that exhaustively combines matching services to achieve a goal.

  Section 4.3 introduces services matching, followed by Section 4.4 which discusses an independent composition algorithm.

- **Integrated and specific composition algorithm.** The algorithm has insight into the internal details of the services, so it is both a matcher and a composer. Capable of solving complex problems, its drawbacks include increased sophistication, decreased performance and no applicability on other classes of task descriptions.

  Example: an algorithm that converts proprietary service descriptions into graph nodes. The resulting graph is searched for a solution path.

  Section 4.5 discusses a powerful integrated composition algorithm.

4.2 Formal definitions

4.2.1 Parameters and variables

When discussing composers, it is convenient to dispose of a formal definition of a service composition. Firstly, we specify sets that appear in the definitions.

- **The set of parameter names** Π which is the union of all possible input and output parameter names of services.
  Examples: *image, name, language*

- **The set of parameter values** Ω which is the union of all possible input and output parameter values of services.
  Examples: `drawing.jpg`, `"George"`, `"en-US"`

- **The set of variables references** Ψ, containing identifiers generated by the composer. These references are used as placeholders for unknown parameter values.
  Examples: `$image1$`, `$name5$`, `$language7$`

4.2.2 Invocations

Definition 4.1 A ***parameter mapping*** β is a function $\beta\colon \Pi \rightarrow \Omega \cup \Psi$ which assigns parameter names to either a value or a variable reference. The set of parameter mappings is B. An element (p, v) of B is called a *parameter assignment* of p to v.

Example 4.1 Parameter mapping with mixed values

Consider the following mapping:

$$\beta_0 := \begin{cases} image & \leftarrow \texttt{drawing.jpg} \\ name & \leftarrow \texttt{\$name5\$} \\ language & \leftarrow \texttt{"en-US"} \end{cases}$$

This means that β_0 is a mapping that assigns the entity `drawing.jpg` to the parameter named $image$, the string `"en-US"` to $language$, and a variable identifier `$name5$` to $name$. A compact notation for β_0 is:

$$\beta_0 := (image = \texttt{drawing.jpg},\ name = \texttt{\$name5\$},\ language = \texttt{"en-US"})$$

Definition 4.2 A ***service invocation*** $I_S(\beta_{in}, \beta_{out})$ is a triple $(S, \beta_{in}, \beta_{out})$ that represents an execution of a service S with input mappings β_{in} and output mappings β_{out}. The domains of β_{in} and β_{out} are the service input and output parameter names, respectively. The parameter value for each parameter name must be an element of the corresponding collaborator parameter domain. The set of all invocations is Φ.

Example 4.2 Book lookup invocation

Consider the `BookLookup` service which looks up the ISBN and DOI information of a book, given the title and author name:

- **Input:** $title$, $author$
- **Output:** $isbn$, doi

Then I_0 represents an invocation of this service:

$$I_0 := I_{\texttt{BookLookup}}((title = \texttt{\$title1\$},\ author = \texttt{"Knuth"}),\ (isbn = \texttt{\$isbn1\$},\ doi = \texttt{\$doi1\$}))$$

This corresponds to an invocation of `BookLookup` for a book whose title is not yet specified, written by Knuth. Because of the similarity to a *method invocation* in programming languages [38], we usually express the above as:

$$I_0 := (isbn = \texttt{\$isbn1\$},\ doi = \texttt{\$doi1\$}) \leftarrow \texttt{BookLookup}\ (title = \texttt{\$title1\$},\ author = \texttt{"Knuth"})$$

4.2.3 Compositions

Definition 4.3 A ***service composition*** C is a directed, acyclic graph with

- a subset Φ_Δ of the invocation set Φ as vertex set;
- a subset Ψ_Δ of the variable reference set Ψ as edge label set.

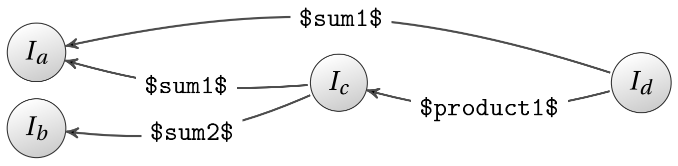
An edge with label ψ from a vertex $I_{S_1}(\beta^1_{in}, \beta^1_{out})$ to a vertex $I_{S_2}(\beta^2_{in}, \beta^2_{out})$ is created if and only if $\psi \in \Psi_\Delta \cap \mathscr{R}(\beta^1_{in}) \cap \mathscr{R}(\beta^2_{out})$. That is: if an input value of the first invocation is a variable reference produced by the second invocation as an output value. An edge between two invocations thus signifies a *dependency* of the first on the second.

In a **complete** composition, all dependencies *resolve* to values or invocation outputs: $complete(C) \Leftrightarrow \forall I_S(\beta_{in}, \beta_{out}) \in \Phi_\Delta : \forall \psi \in \mathscr{R}(\beta_{in}) \cap \Psi_\Delta : \exists I_{S'}(\beta'_{in}, \beta'_{out}) \in \Phi_\Delta : \psi \in \mathscr{R}(\beta'_{out})$. A composition is **partial** if it does not satisfy this requirement.

Example 4.3 Composition of calculus service invocations

Consider the following complete composition of calculus service invocations C_0, which computes the value of the calculation $(1+2)^{(1+2)\cdot(3+4)}$.

$$\begin{cases} I_a := (sum = \texttt{\$sum1\$}) \leftarrow \texttt{Addition}(termA = 1,\ termB = 2) \\ I_b := (sum = \texttt{\$sum2\$}) \leftarrow \texttt{Addition}(termA = 3,\ termB = 4) \\ I_c := (product = \texttt{\$product1\$}) \leftarrow \texttt{Multiplication}(factorA = \texttt{\$sum1\$},\ factorB = \texttt{\$sum2\$}) \\ I_d := (result = \texttt{\$result1\$}) \leftarrow \texttt{Exponentiation}(base = \texttt{\$sum1\$},\ exponent = \texttt{\$product1\$}) \end{cases}$$



Omission of either I_a, I_b, or I_c results in a partial composition.

Definition 4.4 A ***serialized service composition*** $C_\$$ is an ordered list of n invocations $I_{S_i}(\beta^i_{in}, \beta^i_{out})$ that represents an execution plan of a composition C. A serialization of a complete composition satisfies $\forall i \in 1..n : \forall \psi \in \mathscr{R}(\beta^i_{in}) \cap \Psi_\Delta : \exists j \in 1..i-1 : \psi \in \mathscr{R}(\beta^j_{out})$, meaning all dependencies resolve to values or *preceding* invocation outputs.

Example 4.4 Serialized composition of calculus service invocations

Two possible serializations of C_0 in Example 4.3 are:

- $C_{0\$}^a := I_a, I_b, I_c, I_d$: execute "1+2", then "3+4", the product, and the exponentiation;
- $C_{0\$}^b := I_b, I_a, I_c, I_d$: execute "3+4", then "1+2", the product, and the exponentiation.

In the case of C_0, no other order respects the dependencies between the invocations.

4.3 Service matching

4.3.1 Match conditions

The first obstacle in composition creation, is to determine whether two services match. A *start service* S_σ matches an *end service* S_ϵ if an invocation $I_{S_\sigma}(\beta^\sigma_{in}, \beta^\sigma_{out})$ of S_σ exists that enables an invocation $I_{S_\epsilon}(\beta^\epsilon_{in}, \beta^\epsilon_{out})$ of S_ϵ. The first invocation implies fulfillment of both the *input conditions* (necessary to allow the invocation) and the *output conditions* (as a result of the invocation) of S_σ. This signifies that, in order to guarantee a match, the union of the start service's input and output conditions must imply the end service's input conditions.

These input conditions consist of the *input type declarations* and the *preconditions*, as specified in the OWL-S description. Similarly, the output conditions consist of the *output type declarations* and the *postconditions.* In both cases, the type declarations are converted into RDF `type` statements on the input variables, as in the example below.

Note that the output conditions can reference input variables, which is the case for the variable `regionInput` in the example. A correct interpretation of the output conditions consequently requires the availability of the input conditions as well, which is exactly how we approach service matching.

Example 4.5 Output conditions of the Face Recognition Collaborator (Section 3.2)

```
@prefix Process: <http://www.daml.org/services/owl-s/1.1/Process.owl#>.
@prefix Expr: <http://www.daml.org/services/owl-s/1.1/generic/Expression.owl#>.
:FaceOutput Process:parameterType "http://example.org/Faces.owl#Face".
:PersonOutput Process:parameterType "http://example.org/Persons.owl#Person".

:FaceRecognitionEffect Expr:expressionBody
  "?regionInput <http://example.org/Images.owl#regionDepicts> ?faceOutput.
   ?faceOutput <http://example.org/Faces.owl#isFaceOf> ?personOutput.".
```

The above OWL-S output description translates into the conditioned variables below.

```
?regionInput <http://example.org/Images.owl#regionDepicts> ?faceOutput.
?faceOutput a <http://example.org/Faces.owl#Face>;
              <http://example.org/Faces.owl#isFaceOf> ?personOutput.
?personOutput a <http://example.org/Persons.owl#Person>.
```

The example indicates the interweaving of type declarations with preconditions or post-conditions. Indeed, their difference in OWL-S is purely syntactical; type constraints are merely a special kind of conditions. In service matching, we will subsequently consider them together. We will use OWL-S notation and conditioned variable notation interchangeably from this point forward, since they share the same expressiveness.

4.3.2 Combinatorial matching

A naive approach to service matching is to generate all possible combinations of start service variables with end service input variables. We interpret each combination as an assignment. If an assignment exists such that the start conditions satisfy the end service input conditions, the services match. Although this method correctly identifies some matches, it is computationally inefficient and incomplete, as shown hereafter.

4.3.3 Reasoner-based matching

To understand the incompleteness of combinatorial matching, consider the next example.

> **Example 4.6** Finding the format of an image
> Consider a start service and an end service with the following specifications:
>
> - **start service outputs:** `?image a :Image.`
> - **end service inputs:** `?formatName a :FormatName.`

As every image is stored in a certain format, the above services should match. However, no satisfying assignment from start variables to end variables exists. The key here is the existence of a silent intermediary variable – the image format – which relates to the input (every image has a format) and output (every format has a name).

So why not just extend the end service description to mention the relation between image and format explicitly? While this addition would handle the specific case of Example 4.6, it is impossible to account for *all* situations in which this service will be used. For example, an audio file also has a format, and so does a video file; the end service can never anticipate all these variations. Extending the start service in a general way turns out equally impossible: upon the creation of its description, we do not know what image properties will be needed.

The knowledge that every image has a format, is thus clearly separate from the service descriptions. Yet, we need a way to apply this knowledge to the specific instance of the image when required. This kind of complex deductions can be derived efficiently by a *reasoner*. The input and output conditions of the start service are served to the reasoner as assumptions. We add an additional rule with the end service input conditions as antecedent and a `success` message as consequent. To determine whether the services match, we ask the reasoner to infer the `success` message. If this succeeds, the end service rule has been triggered, which in turn indicates that its antecedent is fulfilled, signaling a match.

Essential in the above method is the incorporation of *Semantic Web knowledge* into the reasoner. This practice distinguishes reasoner-based matching from the far simpler combinatorial approach. Knowledge can come in the form of ontologies or rules that express either general or domain-specific facts.

The importance of reusable knowledge

The following knowledge conveys what we *minimally* require to match the services of Example 4.6.

Example 4.7 Specific `Image` knowledge

```
{ ?image a :Image. }
=>
{ _:formatName a :FormatName. }.
```

The above rule expresses that for every image, a format name exists.

Although functionally adequate, this knowledge is very specific to the problem and carries little semantic value. We strive to express knowledge in a way that is as general and reusable as possible, in order to be portable across many different situations. The following knowledge has the same intended effect, but is semantically richer and likely to be reusable.

Example 4.8 Reusable `Image` knowledge

```
:Image rdfs:subClassOf :MediaResource.
:MediaResource rdfs:subClassOf [a owl:Restriction;
                                owl:onProperty :hasFormat;
                                owl:allValuesFrom :Format;
                                owl:minCardinality 1].
:Format rdfs:subClassOf [a owl:Restriction;
                         owl:onProperty :hasFormatName;
                         owl:allValuesFrom :FormatName;
                         owl:minCardinality 1].
```

The above statements are (partial) `Image` and `Format` ontologies with rich semantics. They indicate that all images are media resources, each of which has at least one format, which in turn has at least one name. In contrast to Example 4.7, they formally specify the relation between an image and the name of its format.

This illustrates how *modularity* forms an important aspect of reusable knowledge. It can be achieved by classifying facts at the most general applicable level. Instead of linking format and image directly, we observe that a format belongs to a media resource, and that each image is a media resource.

The reasoner is able to deduct far more information about `Image` and `Format` by combining the above statements with the *RDF*, *RDF Schema*, and *OWL* ontologies and rules, the inclusion of which is generally recommended. For example, if we know that an audio fragment is a media resource, the reasoner can derive this fragment has a format name. If we need the MIME-type of the image later on, a relationship between format and MIME-type is sufficient.

We finish the discussion on service matching and investigate how to employ this technique in a service composer.

4.4 Adjacency matrix composition

4.4.1 Adjacency and extended connectivity matrices

An efficient independent composition algorithm comes into existence when we interpret problem solving as a *path search* from an input state to an output state. Given a matcher to determine the schedulability of two services in succession, we construct a *directed adjacency matrix* of all available services. If we assume that services can only add – not retract – information, it is always possible to reschedule the same service once it has been used, because its preconditions remain fulfilled. This leaves $n \cdot (n-1)$ calculations to create an adjacency matrix of n services.

Definition 4.5 Adjacency matrix of a set of services

An adjacency matrix M of n services $S_1, \ldots, S_n$ is an $n \times n$ binary matrix with

$$M_{ij} = M_{S_i S_j} = \begin{cases} 1 & \text{if } i = j \lor S_i \text{ matches } S_j \\ 0 & \text{if } i \neq j \land S_i \text{ does not match } S_j \end{cases}$$

If m_{ij} is the matching coefficient of S_i and S_j, this leads to the following matrix:

$$M = \begin{array}{c} \\ \mathbf{S_1} \\ \mathbf{S_2} \\ \vdots \\ \mathbf{S_n} \end{array} \begin{array}{c} \begin{array}{cccc} \mathbf{S_1} & \mathbf{S_2} & \ldots & \mathbf{S_n} \end{array} \\ \begin{pmatrix} 1 & m_{12} & \ldots & m_{1n} \\ m_{21} & 1 & \ldots & m_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ m_{n1} & m_{n2} & \ldots & 1 \end{pmatrix} \end{array}$$

An adjacency matrix M enables us to calculate the *extended connectivity matrix* M^∞. An element M^d_{ij} of a classic connectivity matrix M^d determines the number of paths from S_i to S_j with length $d \in \mathbb{N}$. However, we are interested in the existence of *any* path from S_i to S_j, regardless of the number of paths and their length[1].

Definition 4.6 Extended connectivity matrix of a set of services

An extended connectivity matrix M^∞ of n services $S_1, \ldots, S_n$ is an $n \times n$ binary matrix with

$$M^\infty_{ij} = \begin{cases} 1 & \text{if } i = j \lor \text{a path from } S_i \text{ to } S_j \text{ exists} \\ 0 & \text{if } i \neq j \land \text{no path from } S_i \text{ to } S_j \text{ exists} \end{cases}$$

It is equal to the saturating nth power in $\mathbb{B}$ of an adjacency matrix M.

[1]We could use the path length as a first cost estimate of the composition to assist the composition selection process. Yet, this approach consumes too much memory: it requires us to store $n-1$ matrices containing n^2 integers instead of a single matrix with n^2 binary numbers.

4.4.2 Composition creation

Process

The inputs of the composition creation process are a *start service* S_σ and an *end service* S_ϵ. The start service is usually a *source* – meaning it does not have any inputs – and the end service a *sink* – meaning it does not have any outputs. Problems consisting of an *input* and a *request* are transformed into a start service, which produces the input, and an end service, which requires the request.

The algorithm disposes of an internal set of services $S_1, \ldots, S_n$ for use in a composition. Its task is then to find a (possibly empty) list of those services $S_{i_1}, \ldots, S_{i_m}$ such that the list of invocations $I_{S_\sigma}, I_{S_{i_1}}, \ldots, I_{S_{i_m}}, I_{S_\epsilon}$ is the serialization of a complete composition. Parameter mappings are implied by the correspondence of two successive services. If no composition could be found, the algorithm results in an error.

Algorithm

We first test whether S_σ and S_ϵ directly match and return $\emptyset$ if this is the case. If not, we search all services S'_σ and S'_ϵ such that S_σ matches S'_σ and S'_ϵ matches S_ϵ. A complete composition exists if and only if a combination of any of those services (S'_σ, S'_ϵ) exists such that the corresponding element in the extended connectivity matrix $M^\infty_{S'_\sigma S'_\epsilon}$ equals 1. The service list becomes $S_\sigma, S'_\sigma, \Lambda, S'_\epsilon, S_\epsilon$ with Λ a list yet to be determined. If the adjacency matrix indicates that S'_σ matches S'_ϵ, $\Lambda = \emptyset$ and the list is determined, omitting S'_ϵ if it equals S'_σ. Otherwise, we recursively apply the algorithm with $S_\sigma = S'_\sigma$ and $S_\epsilon = S'_\epsilon$. Termination is guaranteed, since a 1 value in the extended connectivity matrix indicates the existence of a path of at most n steps and the length of the undetermined path part decreases in every iteration. The algorithm can be summarized in the following steps (special cases omitted for brevity):

Algorithm 4.1 Adjacency matrix composition

```
Require: $S_\sigma$, $S_\epsilon$, $S = \{S_1, \ldots, S_n\}$, $M$, $M^\infty$, $matcher$
  $S_\Sigma \leftarrow$ all $s'_\sigma$ in $S$ where $matcher.matches(s'_\sigma, S_\sigma)$
  $S_E \leftarrow$ all $s'_\epsilon$ in $S$ where $matcher.matches(S_\epsilon, s'_\epsilon)$
  $(S'_\sigma, S'_\epsilon) \leftarrow$ a $(s'_\sigma, s'_\epsilon) \in S_\Sigma \times S_E$ having $M^\infty_{s'_\sigma s'_\epsilon} = 1$
  $result \leftarrow S_\sigma, S'_\sigma, \Lambda, S'_\epsilon, S_\epsilon$
  if $M_{S'_\sigma S'_\epsilon} = 1$ then
    return $result$ with $\Lambda = \emptyset$, removing repeated list items
  else
    return $result$ with $\Lambda = recurse(S_\sigma = S'_\sigma, S_\epsilon = S'_\epsilon)$
  end if
```

4.4.3 Analysis

Performance

The algorithm performs reasonably fast with limited memory consumption, detailed below.

- **Calculation of adjacency matrix:** the adjacency of n services with $n-1$ services needs to be tested. This operation is costly: $\Theta(S_m n(n-1)) = \Theta(c_m n^2)$ where c_m is the average cost of matching two services, but it only needs to be executed once during the initialization phase. Memory cost is n^2 bits.
- **Calculation of the extended connectivity matrix:** calculating the saturating n^{th} power of an $n \times n$ matrix normally requires n matrix multiplications of $\Theta(n^3)$ each, resulting in $\Theta(n^4)$. However, a binary matrix can be manipulated efficiently using binary AND and OR operations, bringing the total complexity down to $\Theta(c_b n^4 / w)$ with low step cost c_b. Memory cost is n^2 bits during and after the calculation, only performed once. Appendix B contains the computations that lead to these results.
- **Calculation of a path:** The search for matching services for S_σ and S_ϵ takes $\Theta(c_m n)$ each. The existence of a path is verified in constant time. Reconstruction takes at most n steps. In each subsequent step, matching services need not to be found by $matcher.matches$ but to be searched inside the adjacency matrix, costing $\Theta(c_b n)$. This brings the total execution time to $\Theta((c_m + c_b) \cdot n)$.

The number of services n is likely to be of a magnitude similar to that of the processor word size w, so the n/w factor will be negligible in practice.

Application range

The adjacency composer can only solve a limited class of problems, the so-called composition chains where each service only uses inputs produced by the previous one. Real-world applications demand more general compositions, in which each service has access to all produced inputs. Enhancements to overcome some of these limitations are possible:

- **Subdivide the end service** and create different compositions for each of its input parameters, that are joined afterwards. This effectively removes the chaining limitation from the last service. Extension of this principle to internal services is impossible, since this is equivalent with exhaustive composition, a practice which ultimately results in combinatorial explosion.
- **Combine internal services with overlapping input parameters** and compose them at once. Since the number of overlapping parameters will be small compared to the number of services, the number of combinations remains reasonable.

None of these enhancements enable a holistic view on the composition, which is necessary for complex problems as described in the next section.

4.5 Reasoner-based composition

4.5.1 Holistic problem solving

Consider the following composition problem example, which appears trivial on first sight.

Example 4.9 Problem *FindImageColorNames*

The goal is to find the names of all colors that appear in a given image.
The image `drawing.jpg` contains two colors: red (`#FF0000`) and yellow (`#FFFF00`).

- **Input:** `<drawing.jpg> a :Image.`
- **Request:** `<drawing.jpg> :contains ?color. ?color :hasName ?colorName.`
- **Available services:**
  - ***ColorDetector***
    - ◇ **Input:** `?image a :Image.`
    - ◇ **Output:** `?image :contains ?color. ?color :hasValue ?colorValue.`
  - ***ColorGuide***
    - ◇ **Input:** `?lookupColor :hasValue ?value.`
    - ◇ **Output:** `?lookupColor :hasName ?name.`

We might be tempted to think that the solution is a simple chained composition, which an adjacency composer could solve. This is not the case: the input directly matches the *ColorDetector* service, yet the *ColorGuide* service does not fully match the request. The twofold request cannot be handled because, although a service for images and a service for color names exist, neither of these is aware of *both* images and color names.

Splitting the request into separate parts and then solving those independently, does not correctly address the request either:

1. "`<drawing.jpg> :contains ?color.`" is **answered as expected**;
2. "`?color :hasName ?name.`" by contrast, **behaves unpredictably**, as `color` could refer to *any* color in the universe. The `name` output could be bound to "pink" or "orange", even though no such color is actually present in the image.

The problem manifests itself because the solution method lacks a global vision on the request. Even a simple example such as the one above, confronts us with the powerful effects of *holism:* a succession of services is able to generate more information than solely the aggregation of the information generated by the individual services. This illustrates at the same time the far-reaching possibilities of compositions and the difficulty of harnessing their power. It has become clear that we need a new class of problem solving algorithms to handle complex requests.

4.5.2 Process

Algorithms with a separate matcher can employ a reasoner to determine the compatibility of two OWL-S descriptions. Reasoner-based composition takes this idea a step further and relies on the reasoner for the composition itself using the following process:

1. each service is **translated** into an N3 rule that encloses its functionality;
2. a reasoner determines whether `request` can be **deduced** from `input`;
3. corresponding compositions are **reconstructed** from the rules used for deduction.

Obviously, the first action executes only once. The number of deductions found in step 2 indicates how many possible compositions exist. In contrast to the algorithm of Section 4.4, all possible solutions are generated simultaneously due to the way reasoners work. The individual actions are discussed in the next subsections.

4.5.3 Translation into N3 rules

Based on an OWL-S description, an N3 rule is created, simulating the execution of an actual service. Instead of producing actual content, the rule creates placeholders. The conversion process translates input conditions into antecedents and output conditions into consequents. Input parameters become unbound variables, outputs parameters become placeholder variables that will be instantiated upon execution of the rule.

Example 4.10 N3 rule translation of the *ColorDetector* from *FindImageColorNames*

```
{
  ?image a :Image.
}
=>
{
  ?image :contains ?color.
  ?color :hasValue ?colorValue.
}.
```

For each image present in the input, the above rule will generate different anonymous nodes to represent `color` and `colorValue`[2].

Next, we need to complement the rule with descriptive information that tells us about the service invocation: which service was invoked, using which input and output parameters? This tracking information will enable us to reconstruct the composition later on.

[2]One may wonder why the consequent contains variables rather than anonymous nodes. This is because the latter have limited scoping (only a single RDF context) which would be insufficient for complex output conditions containing reifications. By using a variable, we ensure ourselves that it will be bound in the entire scope of the consequent.

Therefore, we add to the consequence of the rule a `boundBy` statement, with the output mapping as subject and the service name and input mapping as object. We divert from the triple notation of invocations and use method call notation instead, to emphasize the action of the rule. The parameter assignments of the input and output mapping are formatted as a list of `isAssigned` statements, meaning the subject parameter is assigned the object value. Below is an example of tracking statements, using a special namespace for composer-related entities to avoid interference with request or collaborator entities.

Example 4.11 N3 rule translation of *ColorDetector*, adding tracking statements

```
@prefix c: <composer://composer/>.
{ ?image a :Image. }
=>
{
  ?image :contains ?color.
  ?color :hasValue ?colorValue.
  ({<color> c:isAssigned ?color.}
   {<colorValue> c:isAssigned ?colorValue.})
     c:boundBy [a c:Invocation;
                c:hasService <ColorDetector>;
                c:hasInput ({<image> c:isAssigned ?image.})].
}.
```

Note that some reasoners have an option to display a proof of the deducted knowledge, eliminating the need of tracking. However, such a proof contains a lot of unnecessary details and is more difficult to interpret than our custom tracking statements.

4.5.4 Reasoner deduction

Now that we dispose of N3 rules for all services, we need one more rule representing the request. Again, information to track the binding is added, using a `hasBinding` statement.

Example 4.12 N3 rule translation of the *FindImageColorNames* request

```
@prefix c: <composer://composer/>.
{ ?image :contains ?color.
  ?color :hasValue ?colorValue. }
=>
{ _:solution c:hasBinding ({<image> c:isAssigned ?image.}
                           {<color> c:isAssigned ?color.}
                           {<colorName> c:isAssigned ?colorName.}). }.
```

The reasoner is called with the service rules, request rule, and input statements. We ask to deduce all possible `boundBy`, `hasBinding` and `isAssigned` statements, which are then stored for composition reconstruction.

Reasoner operation

The reasoner will attempt to create `hasBinding` statements and thus try to use the request rule, as this is the only way to generate such statements. This requires the fulfillment of the rule's antecedents, each of which can be satisfied either directly by the inputs or by a collaborator rule. In the latter case, the fulfillment of the rule's antecedents is necessary, again by inputs or a rule. The output is built up recursively using this principle.

It is possible to add custom ontologies and rules to the reasoning process, analogous to their usage with service matchers as described in Section 4.3. However, their application here enables even more powerful service compositions, because their activity span is not limited to two successive service invocations but applies to the composition as a whole.

The way rules are processed, results in a holistic view on the composing process and exposes the versatility of this algorithm.

Output interpretation

The reasoner output can be interpreted in two directions: either from the *reasoner viewpoint* (from the solution binding towards the inputs) or from the *composition execution viewpoint* (from the inputs towards the solution). We desire the latter interpretation, yet this requires some additional processing, carried out in the reconstruction step.

Below is an example of possible reasoner output. Generated variables names, representing value placeholders, are distinguishable by surrounding dollar signs.

Example 4.13 Possible reasoner output of the color name search

```
@prefix c: <composer://composer/>.
_:solution1 c:hasBinding ({<color> c:isAssigned <$color1$>.}
                          {<colorName> c:isAssigned <$name1$>.}).

({<name> c:isAssigned <$name1$>.})
   c:boundBy _:invocation1.
_:invocation1 a c:Invocation;
              c:hasService <ColorGuide>;
              c:hasInput ({<lookupColor> c:isAssigned <$color1$>.}
                          {<value> c:isAssigned <$colorValue1$>.}).

({<color> c:isAssigned <$color1$>.}
 {<colorValue> c:isAssigned <$colorValue1$>.})
   c:boundBy _:invocation2.
_:invocation2 a c:Invocation;
              c:hasService <ColorDetector>;
              c:hasInput ({<image> c:isAssigned <drawing.jpg>.}).
```

4.5.5 Composition reconstruction

When the reasoner's work has been done, we transform the generated statements using a three-step process:

1. find all **solution bindings**, indicated by `hasBinding` statements;
2. recursively trace all **variable mappings** to generate the invocation dependency graph;
3. linearize the composition to obtain a **serialized composition** of invocations.

These steps are displayed in the algorithm below.

Algorithm 4.2 Composition reconstruction

```
Require: $reasonerStatements$
  $serializedCompositions \leftarrow \emptyset$
  $boundBy \leftarrow reasonerStatements$ where predicate = boundBy
  $solutionBindings \leftarrow reasonerStatements$ where predicate = hasBinding
  for all $solutionBinding \in solutionBindings$ do
    $composition \leftarrow \{new$ EndService($solutionBinding$)$\}$
    repeat
      $bindingInvocations \leftarrow$ invocations binding any parameter of $composition$,
                                          found through lookup in $boundBy$
      $composition \leftarrow composition \cup bindingServices$
    until $composition$ does not change
    $serializedComposition \leftarrow [\,]$
    repeat
      $availableInvocations \leftarrow invocation$ from $composition$
                                          whose dependencies are in $serializedComposition$
      $serializedComposition \leftarrow serializedComposition, availableInvocations$
    until $serializedComposition$ does not change
  end for
  return $serializedCompositions$
```

The algorithm produces the correct result, because of the following reasons:

- Each `hasBinding` statement corresponds with exactly one possible composition. The only rule that creates such a statement is the request rule, which can solely be triggered if the solution bindings were successful.
- Each `boundBy` statement uniquely identifies the invocation that executed the binding, because those statements are only created by service rules, which can solely be triggered if their input conditions are satisfied.
- *availableInvocations* will not become empty until the composition is finished: because the composition exists, a path that respects dependencies must exist as well.

Necessity of reconstruction

At first, one may wonder why an additional reconstruction step is needed. If the reasoner can track all service invocations, can it not also track the whole composition? While possible, this strategy is not recommended. Reasoners typically function in a goal-driven way; their *control flow* goes backwards. Applied to service matching, this means a reasoner begins with the end service and work towards the start service. On the contrary, the *data flow* goes forward, so when the start service parameter data is available, the end service data is not known. This explains why it is easier to reconstruct the composition afterwards, when the data flow can be read in reverse, in parallel with the control flow. We then start with the last invocation, whose data is now available, and work towards the first invocation.

4.5.6 Complex service descriptions

The usage of service as N3 rules described here, silently assumes several simplifications. For example, OWL-S descriptions can contain services with multiple precondition and result sets, each of which only applies on certain conditions. We view all these possibilities as separate services for simplicity. Also, composite services and erroneous results were not taken into account. A method for handling multiple preconditions and results of OWL-S services is described in [37], which details translation into SWRL rules.

4.5.7 Analysis

Performance

The algorithm's computational complexity is difficult to express, due to the many parameters involved in the calculation such as the number of statements and different predicates. It suffices to say that even complex compositions using several ontologies and rule sets, can be performed in an acceptable amount of time.

Application range

Reasoner-based composition is able to solve highly sophisticated problems, thanks to the intelligent application of Semantic Web knowledge on the problem as a whole. In theory, any composition problem can be solved, given sufficient knowledge. This is both the biggest advantage and disadvantage: the reasoner depends on available knowledge resources. If the application domain is known in advance, special care should be taken to select appropriate sources or to create domain-specific knowledge. Chapter 7 provides a detailed example.

4.6 Extensions

The concept of service composition can be extended with additional capabilities that extend its possibilities substantially. The following extensions are described for reasoner-based composers, but their concept is portable to other types of composers as well.

4.6.1 Partial compositions

If no complete composition could be found, it may be worthwhile to test whether partial compositions can be derived. A partial composition has at least one unbound input. We could attempt to solve as many parts of the solution as feasible, possibly completing the composition later on if certain services generate additional information.

A straightforward implementation is to supply the reasoner with additional statements that fulfill the antecedents of the reasoner rules using dummy values. We then proceed as usual with the composing process. The reasoner output can contain compositions with invocations that have dummy values as input parameters, which need to be replaced by unbound variable placeholders. The result needs to be filtered to remove possible empty and duplicate compositions introduced by the dummy values.

4.6.2 Helper services

Services with list outputs

The critical reader may have noticed that the *ColorDetector* of Example 4.9 in Section 4.5 does not rigorously indicate that it returns multiple colors. A better way to formalize the output of this collaborator is listed below.

Example 4.14 *ColorDetector* output with a list of colors

```
?colorList a rdf:List.
_:elementClass owl:oneOf ?colorList;
               rdfs:subClassOf :Color;
               rdfs:subClassOf [a owl:Restriction;
                                  owl:onProperty :appearsIn;
                                  owl:hasValue ?image].
```

The above statements demand that each item of `colorList` is a color contained in the input `image`, though this may not appear self-evident. The `owl:oneOf` statement requires each list item to be an instance of the class `_:elementClass`. In turn, this anonymous class has restrictions from which the desired item properties follow.

The issue now is how to match this service to `ColorGuide`, which accepts a single color instead of a list of colors. As a first attempt, we could create a rule that assumes the existence of at least a single element.

Example 4.15 N3 rule assuming restricted lists contain an element

```
{ ?class owl:oneOf [a rdf:List]. } => { _:element a ?class. }.
```

This rule will allow the creation of the composition. However, one of its variable references will be unbound as depicted below.

Example 4.16 Resulting composition using the N3 rule for lists

1. ($colorList$ = `$colorList1$`) ← `ColorDetector` ($image$ = `drawing.jpg`)
2. ($name$ = `$name1$`) ← `ColorGuide` ($color$ = `$element1$`)

The above composition appears to be correct since it is clear to us that `$element1$` is the generated identifier for an element of `$colorList1$`. Yet, this semantic relationship is not indicated in the invocation list, which will result in the interpretation of `$element1$` as an unbound variable, rendering the composition partial.

The solution to this problem is a *helper service* that handles lists. This is a virtual service used as a placeholder during the composing process, which gets replaced by a specific task after composition reconstruction. Instead of the rule of Example 4.15, we include a list helper service which executes the same rule. The difference is that this service will be temporarily included in the composition, yielding the following output.

Example 4.17 Resulting composition using the list helper service

1. ($colorList$ = `$colorList1$`) ← `ColorDetector` ($image$ = `drawing.jpg`)
2. ($element$ = `$element1$`) ← `ListHelper` ($list$ = `$colorList1$`)
3. ($name$ = `$name1$`) ← `ColorGuide` ($color$ = `$element1$`)

After composition reconstruction, we remove the virtual `ListHelper` invocation, changing the `$colorList1$` placeholder into an *annotated variable reference* to indicate its *“list of”* relationship with `$element1$`. The entity executing the composition can then opt to execute `ColorGuide` for each element in the list generated by the `ColorDetector` invocation.

Other purposes

Other uses of helper services may include the instantiation of subproperties or the calculation of expression results such as mathematical expressions or string operations. Helper services allow even more flexibility in creating compositions, at the cost of more complex variable placeholders which need to be understood by the composition executor.

Chapter 5

Supervision

Having looked at collaborators and their compositions, we zoom out and direct our attention to the supervision of the problem solving process. The plan generated by a composer is turned into an actual execution which aims to generate a solution.

5.1 Definition

First, we define a process supervisor and identify its role in the problem solving system.

> **Definition 5.1** A ***supervisor*** is a component responsible for solving a problem using collaborators and an execution plan composer. Its tasks include:
>
> 1. **selecting** the appropriate execution plan (Section 5.2);
> 2. **executing** this plan (Section 5.3);
> 3. **displaying** the solution process progress (Section 5.4);
> 4. **recovering** from unanticipated output or errors (Section 5.5);
> 5. **formulating** a response to the request (Section 5.6).

The problem solving process can start when the following items are in place:

- a set of **collaborators** accompanied by a semantic description;
- a **composition algorithm** with domain-specific ontologies and rules;
- a **blackboard** for information storage and organization;
- **semantic knowledge** to derive new information;
- a **request** with a certain **input**.

The supervising process is governed in a semantic way, using Semantic Web technologies. This enables a supervisor to solve complex problems while being able to explain every step in a human-readable way. As this dissertation focuses on the problem solving method itself rather than the solutions it produces, this behavior is an important design goal. The next few sections describe the various supervisor tasks listed above in more detail.

5.2 Composition selection

The supervisor firstly demands the composer to search for *complete compositions*. If none were found, *partial compositions* can also be considered. The compositions are then evaluated by criteria such as the following, which should be balanced against each other.

- **Cost:** the expected cost associated with the execution of the collaborators. This cost is at least the sum of the individual execution costs, but it can increase in case of failure. It should be expressed as a mixture of different quantities, such as processor time and amount of money, as external services and employees can be involved.
- **Accuracy:** some collaborators have a higher success rate than others, usually at the expense of a higher cost.
- **Performance:** faster compositions should be allocated to urgent tasks.
- **Availability:** some collaborators are not always available, which can be due to server outage or working schedules if the collaborator task involves people.
- **Completeness:** if the request cannot be solved entirely or if the proposed solution is too expensive, other solutions that only solve part of the problem can be included.

The problem described here is that of *constraint-based* composition selection and optimization. Simple cases, such as cost minimization, can be solved using traditional shortest path algorithms. Clearly, in more complex cases, an absolute optimum does not exist but rather corresponds to a *compromise between trade-offs* of several criteria. It is also possible for the composer to reckon with request constraints, by sorting compositions according to suitability or by aborting a composition when parameters reach a specified threshold.

Often, the selection process is hindered by the *inaccuracy of the parameter estimates*, which strongly depends on the received input. Several collaborators can perform the same task using a different algorithm, each of which is specialized on certain problem properties. If these properties are unknown in advance, we cannot decide meaningfully. It could prove profitable to execute part of the composition until at least some properties are known, followed by a recomposition using the newly available knowledge.

Other considerations could be taken into account. In case of *partial composition* for example, we might be interested to push invocations with unbound variables to the back, trying to do as much work as possible. When we reach such an invocation step during execution, we cannot proceed directly. Instead, we check if any previous invocation perhaps resulted in additional information, either directly or indirectly by the use of semantic knowledge. Again, we recompose using the new information and try to continue.

We have chosen not to pursue the path of composition optimization in this dissertation, and would like to refer to Chapter 8 for an overview of future possibilities. As our focus is on the problem solving process itself, we limit composition selection to picking any composition that matches the current execution state and satisfies the requirements.

5.3 Composition execution

5.3.1 Algorithm

When the supervisor receives a new request, it initializes the blackboard and adds the input. The composer transforms the blackboard and the request into in a number of possible executions, the best of which is selected. We have to keep track of these additional items:

- the **current information** which is kept on the blackboard;
- the **variable binding**, a mapping of variable identifiers and values;
- the **current composition**, **current invocation** and all **past invocations** with results.

Since a composition consists of an invocation list, its execution – in the most basic form – comes down to the execution of these invocations. This algorithm is detailed below.

Algorithm 5.1 Execution of an invocation list

```
Require: $invocations, binding, blackboard$
  for all $invocation \in invocations$ do
    $collaborator \leftarrow invocation.collaborator$
    $inputs \leftarrow invocation.inputMapping$, replacing variables using $binding$
    $output \leftarrow collaborator.execute(inputs)$
    $blackboard \leftarrow blackboard \cup output.statements$
    $binding \leftarrow binding \cup invocation.outputMapping$, replacing variables using $output$
  end for
  return $blackboard$
```

The replacement steps of the algorithm are best illustrated by an example that traces through a single invocation step.

Example 5.1 Single invocation execution; translation of a phrase

State before invocation

- **blackboard:** `<speech.txt> a :Text; :containsPhrase <PhraseA>.`
- **binding:** *(text1* = `speech.txt`*, phrase1* = `PhraseA`*)*
- **invocation:** *(translation = translation1)* ← `Translator`*(original = phrase1, lang =* `"fr"`*)*

State after invocation

- **input:** *(original* = `PhraseA`*, lang* = `"fr"`*)*
- **output:** *(translation* = `PhraseB`*)*
- **binding:** *(text1* = `speech.txt`*, phrase1* = `PhraseA`*, translation1* = `PhraseB`*)*
- **blackboard:**
```
<speech.txt> a :Text; :containsPhrase <PhraseA>.
<PhraseA> :hasTranslation <PhraseB>.
```

The execution of `Translator` returns `Phrase2` as value for the $translation$ parameter. The invocation specifies that this value should be assigned to the variable *translation1*.

5.3.2 Variable binding

The variable binding clearly plays a crucial part in the contiguity of the execution and deserves some explanation. Its concept is similar to that of a *single-assignment store* [38, p. 42] in programming languages, meaning that once a variable is assigned to, its value cannot change. The following definition is analogous to Definition 4.1 of a parameter mapping.

> **Definition 5.2** A ***variable binding*** β_v is a function $\beta_v\colon \Pi_v \to \Omega_v$, which assigns variable names to a simple value ($\in \Omega \cap \Omega_v$) or complex value ($\in \Omega_v$). The set of all bindings is B_v. An element (n, v) of B is called a *variable assignment*, assigning v to n.

As demonstrated, the variable binding keeps track of the result variables and their reuse as input variables. We therefore can consider each collaborator invocation as an external method invocation. The composition is in fact a *declarative program* [38] whose execution order is solely governed by *data dependencies*. This declarativeness follows naturally from the fact that a composer constructs a plan that indicates *how* to solve a certain problem. In contrast, the supervisor *interprets* the declarative program, determining *what* steps should be performed. The variable binding is the entity that connects both views. We can take advantage of this high level of freedom to exploit parallel or batch execution capabilities.

Resource values

In its most elementary form, a variable will be assigned to an RDF resource of the blackboard. The variable effectively functions as a reference to more detailed information about the resource on the blackboard. The difference lies in the fact that the variable binding is a process artifact only relevant during the execution, whereas the blackboard contains persistent information.

Complex values

In Subsection 4.6.2, we introduced *annotated variable references* to model more complex variable relationships. These correspond to complex values in Ω_v.

> **Example 5.2** Multi-valued variable assignments
>
> Continuing Example 4.17 of Subsection 4.6.2, solely storing the value of `$colorList1$` does not convey the intended meaning, as this value is just a list identifier. The annotation on its variable reference indicates that elements of this list are referred to by `$element1$`. So in addition to the `$colorList1$` assignment, we also add a multi-valued assignment of the elements of the list to `$element1$`. This translates the `ColorGuide` step of the original composition into invocations for *each* value of `$element1$`.

5.4 Progress display

5.4.1 Purpose

An insight into the progress of the execution is crucial for research purposes as well as practical applications. If a request cannot be fulfilled completely, or if the supervisor was not able to find a solution within the specified restrictive criteria, we should be able to investigate what went wrong. We then can examine which additional knowledge sources or collaborators could assist the supervisor to solve similar problems in the future.

5.4.2 Chronological execution display

Executions of simple compositions with relatively little invocation branches, can be displayed as a *chronological list of process steps*. It provides the user with a detailed overview of each invocation and its parameters. In addition, status messages – reporting activities such as semantic enrichment of the blackboard – can be included in this list.

Although this view is concise and convenient for small compositions, the intentions of the supervisor and the direction it is heading to are not obvious at every point in time.

5.4.3 Hierarchical, goal-directed display

A composition is best understood from goals towards inputs (*"to find y, we need x"*), yet execution proceeds in the reverse direction (*"find x in order to find y"*). A *hierarchic, goal-directed representation* is an efficient way to describe the current execution state. The composition itself does not form a strict hierarchy, but every output node is the root of a tree.

The result is a display similar to the one below. Additionally, each invocation node contains its input and output parameters, the result of the execution and – in case of failure – a description of the fault and the alternative path.

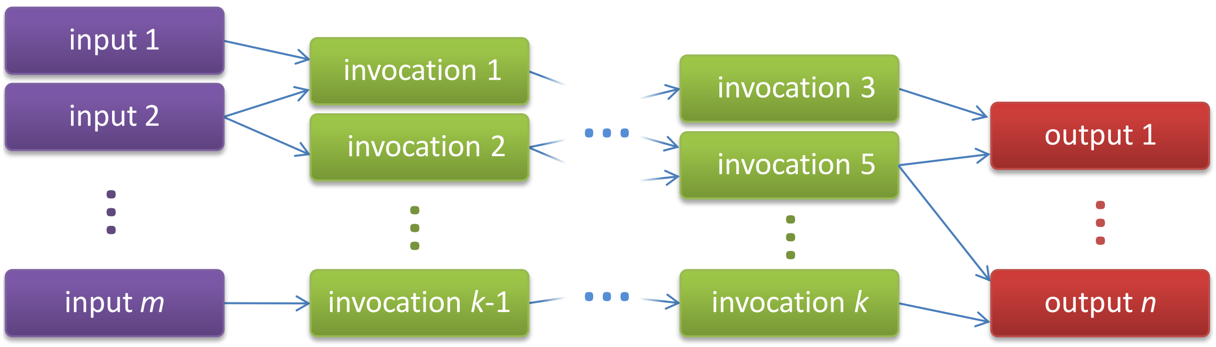


Figure 5.1: Hierarchical, goal-directed display of a composition execution

The chronological and hierarchical views can be combined to provide an image of both the instantaneous progress and the process as a whole.

5.5 Failure recovery

5.5.1 Process

The composer optimistically assumes correct and successful behavior of all involved collaborators. If we were to withdraw this assumption, the construction of viable compositions would be virtually impossible since every collaborator can be subject to failure. The supervisor therefore handles error recovery, a process consisting of:

1. **failure detection:** catching errors and incomplete output;
2. **impact determination:** defining the consequences for the rest of the execution;
3. **plan adaptation:** change the plan to reach (possibly adjusted) goals in a different way.

These substeps are detailed in the next few sections.

5.5.2 Failure detection

We distinguish two kinds of failures: *errors* during collaborator execution and normal execution with *incomplete output*. Since the surrounding programming environment usually detects errors by an exception mechanism, we assume that this task is trivial. To detect incomplete or empty input, we make use of the invocation's output mapping. If certain parameters of the output mapping do not appear in the output, it is incomplete and we should initiate the failure recovery process.

5.5.3 Impact determination

Once the *point of failure* is identified, we can determine the failure impact by searching for the invocations that – directly or indirectly – depend on its output. At least one invocation will be affected, since only outputs necessary for future invocations are mapped. The failure repeatedly propagates through these invocations, eventually reaching one or several of the solution generating invocations. The *affected part* of the composition consists of all these invocations, starting at the point of failure. The affected part in Figure 5.2 spans the failed invocation and the two rightmost invocations.

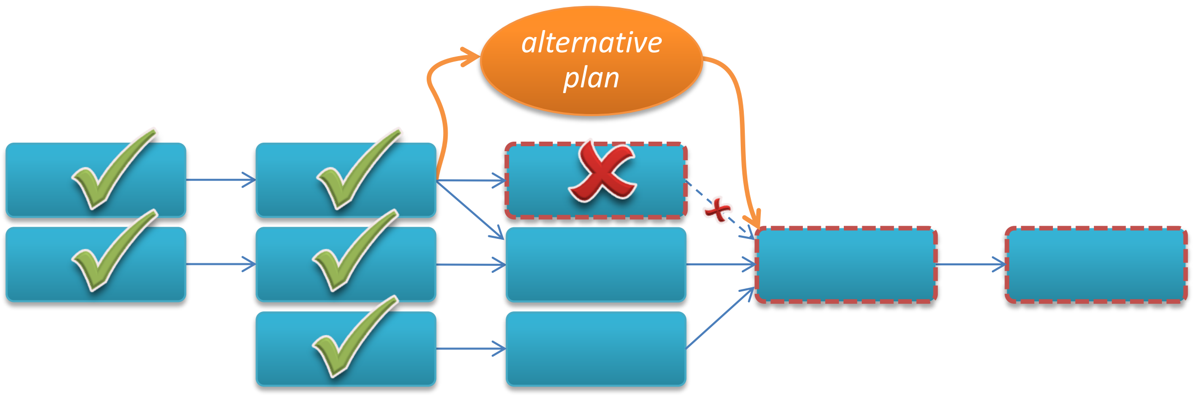
The severity of the impact indicates whether the composition should be adapted locally or recreated as a whole. We designate a *resumption point* where normal execution is continued. Figure 5.2 shows the same failure with different resumption points. In Figure 5.2(a), this is the second invocation from the right; in Figure 5.2(b), it is the rightmost invocation. The selection of the resumption point is influenced by the availability of an alternative plan and the history of attempted invocations.

5.5.4 Plan adaptation

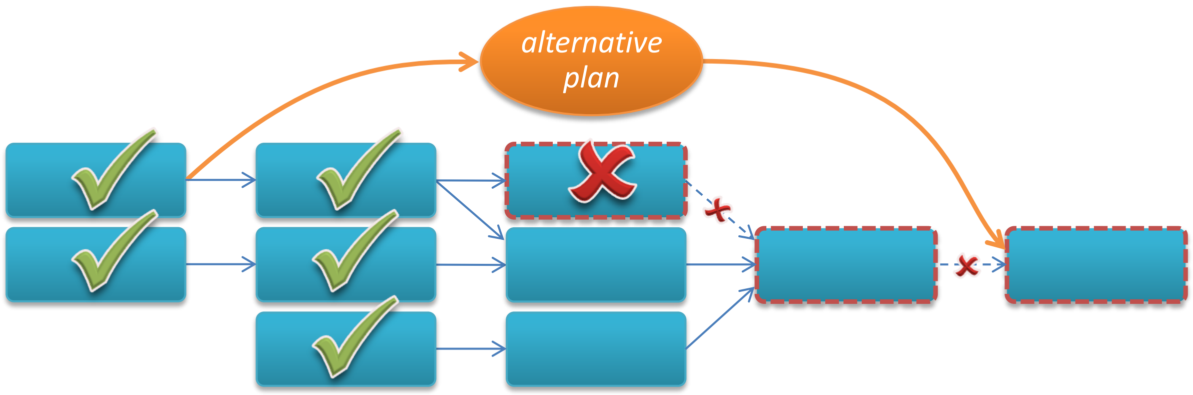
To recover from failure, the supervisor asks the composer to generate compositions for the affected part of the plan. New compositions start with the current state of the blackboard and end in the resumption point. Prior to the generation, the supervisor deduces as many additional facts as possible from the blackboard using Semantic Web knowledge. The amount of available information is generally larger than that at the time of the initial composition, since the partial execution may have yielded intermediary results. As a result, new compositions that make use of this increased knowledge are possible.

We only consider compositions without previously executed invocations – failed or successful – to avoid infinite failure recovery loops and the overhead of duplicate invocations, whose results are already known. Filtering at this level is necessary, since the composer itself is unaware of any failures and would otherwise suggest the same solutions over and over.

The figure below shows two different strategies to handle the same failure. Figure 5.2(a) depicts an adaptation which tries to correct the failure locally, whereas Figure 5.2(b) uses an entirely new composition. Of course, for more complex compositions, various intermediate degrees exist. The supervisor decides what strategy to use, based on the problem parameters, the failure point, and the possible history of failed recovery attempts.



(a) Plan adaptation with local recovery; affected part highlighted



(b) Plan adaptation with global recovery; affected part highlighted

Figure 5.2: Different plan adaptation strategies

5.6 Response output

At the end of the supervision process, the blackboard contains all RDF statements related to the solution. The variable binding indicates which RDF resources are bound to the output parameters. A brief response consists only of the requested output resources, but often certain intermediary results are of interest as well. This leaves us with three possibilities for response output.

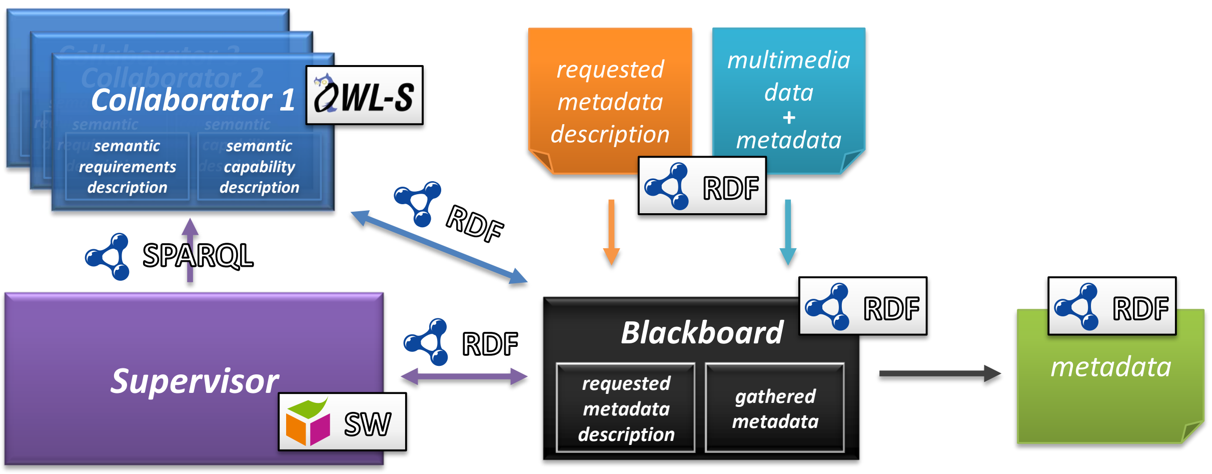
- **Requested output only:** the request parameters are bound using the variable binding and returned. All other information is discarded.
- **Requested output and statements:** in addition to the response output, the supervisor also returns the blackboard statements.
- **Blackboard output only:** the request is considered as a guideline directing the search. The semantic relationships between the blackboard contents, containing the inputs and the output variables, enclose the requested information.

For *metadata generation applications*, this last option proves especially interesting. They constitute an example of requests that merely hint what a typical solution could look like, leaving room for other discovered data instead of demanding a predefined solution.

Note that the RDF output can be translated into other formats that better fit the problem and solution domain. Input parameters can be adapted similarly.

5.7 Final system overview

The discussion of the supervisor component finishes the description of the entire semantic problem solving system. We now revise the architecture as displayed in Figure 2.2 of Chapter 2, adding technology mappings. This system differs from others in that each process step retains the semantics of the problem and the solution, as depicted in the diagram below.



Collaborator 1
OWL-S
semantic requirements description
semantic capability description
requested metadata description
multimedia data + metadata
RDF
RDF
SPARQL
RDF
Blackboard
RDF
requested metadata description
gathered metadata
RDF
metadata
Supervisor
SW

Figure 5.3: Final semantic problem solving system architecture

Chapter 6

Implementation

The concepts elaborated in the previous chapters have been implemented into a software platform to demonstrate practical applications. This platform closely follows the theoretical foundations laid down earlier.

6.1 Platform

The platform is intended both as a research tool and as a software library for real-world applications. Dubbed by the acronym *Arseco*, which stands for *Arcade of Semantic Collaborators*, it consists of:

- **collaborator libraries**: modules that facilitate common collaborator tasks, such as SPARQL query parsing and back-and-forth translation of RDF;
- a **semantic problem solving engine**: an independent component that is able to solve complex problems using collaborators.

Two important points need to be mentioned: firstly, collaborators are regular SPARQL endpoints that can be created without the use of the assisting libraries. Secondly, the problem solving engine functions on a high level and is not specific in any way to the problem under consideration. Solving domain-specific problems is achieved by using specialized collaborators and knowledge. We refer to Chapter 7 for the discussion of such a use case.

We identify the following main architectural concerns for the Arseco platform:

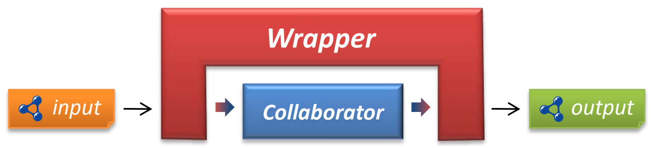
- **modifiability**: we must be able to exchange components easily, to research the impact of certain changes and to adapt quickly to different problem situations;
- **interoperability**: collaborators with different implementations should be able to work together transparently without additional adaptations;
- **reusability**: components should be as independent as possible from specific problems, maximizing their reuse.

The following sections detail the technical structure of the implementation, linking theoretical concepts to actual software components.

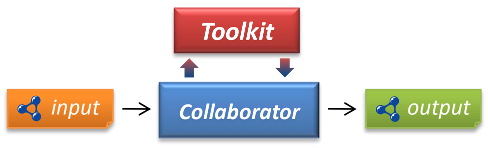
6.2 Collaborators

6.2.1 Framework

The only demand in a collaborator contract is that it executes a well-defined task, accepting SPARQL as input and generating RDF as output. The specification does not impose any limits on programming language, platform, or other technical aspects. This great amount of flexibility renders it unfeasible to create a framework that encompasses all possible scenarios. Instead, we should strive to cover the most common cases. Two major approaches exist, as depicted in Figure 6.1.



(a) The collaborator uses the library as a wrapper



(b) The collaborator uses the library as a toolkit

Figure 6.1: Different approaches to the collaborator framework

Wrapper approach

In the *wrapper* approach (Figure 6.1(a)), the framework acts as an intermediary between the external world and the collaborator, similar to Algorithm 3.1 in Subsection 3.1.3. One end of the wrapper translates the input inside the SPARQL-query into an internal input representation understood by the collaborator, which is then executed. After execution, the other end converts the internal output format of the collaborator into RDF statements, against which the query is subsequently performed.

The image clearly displays the advantage of wrappers: existing collaborators can be incorporated without any internal modification. This also implies a major disadvantage: each collaborator needs a different wrapper adaptation. While generalized wrappers can be created, they require custom configuration to fit a specific collaborator.

Toolkit approach

The collaborator itself takes control of the process in the *toolkit* approach (Figure 6.1(b)). The library offers a set of functions the collaborator can access, such as RDF parsing and SPARQL processing. The advantages of this approach lie in increased efficiency – since no translation is necessary – and the absence of custom configuration. This comes at the cost of modifications to the collaborator to communicate with the framework. Of course, new collaborators do not introduce this overhead.

To summarize, we can say that the wrapper method works best for existing collaborators whereas the toolkit method excels with collaborators under development.

Implemented software

To ensure compatibility with a wide range of software, C++ and a C# implementations of the collaborator library have been created, named *ArsecoCommon* and *ArsecoCommonSharp*, respectively. They both share the class diagram depicted in Figure 6.2.

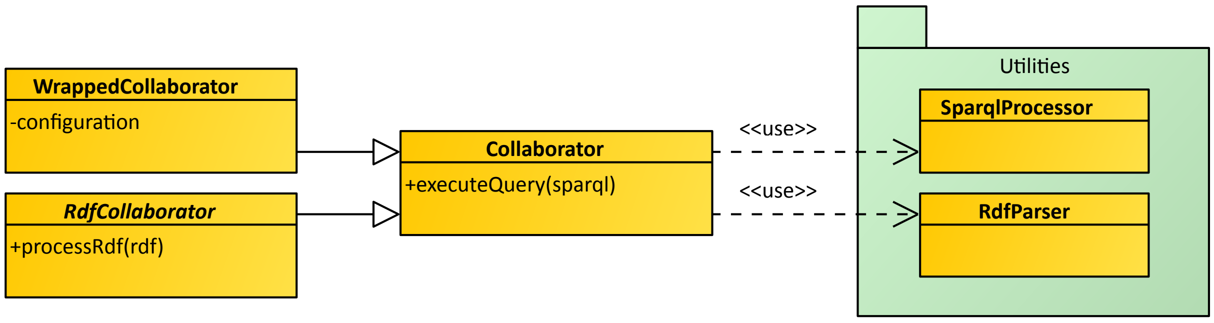


Figure 6.2: Class diagram of the collaborator framework

`Collaborator` is an abstract base class for collaborators. `WrappedCollaborator` can be instantiated with the right configuration parameter for the wrapper approach. The toolkit approach can be implemented conveniently by inheriting from the `Collaborator` class, or by using the classes in the `Utilities` namespace directly. Alternatively, one can choose to inherit from the `RdfCollaborator` class, which abstracts the SPARQL part and only requires the implementation of an RDF processing method.

Software that cannot interface with C++ or C#, can still use the toolkit approach by invoking a standalone command line version of ArsecoCommon. This portable program supports several RDF and SPARQL related commands.

6.2.2 Server

To provide a convenient means of accessing collaborators, I created a basic HTTP web server program that houses several collaborators. This server, whose implementation is a purely technical matter, offers the following functionality:

- a **configuration-based** way to add and remove collaborators;
- a **list of all available collaborators** in HTML, XML, and RDF format;
- for each collaborator
  - an implementation of the **SPARQL protocol** HTTP bindings [11];
  - **OWL-S description** access;
  - an **HTML form** to enable browser-based execution.

The use of an HTTP server forms a key abstraction towards the solution engine. We preconfigure the URIs of the XML collaborator lists, which the engine then loads dynamically. These lists provide the locations of the OWL-S descriptions, from which all required information for composition and execution can be loaded.

6.3 Engine

6.3.1 Design decisions

Portability is not as important for the engine as it is for collaborators, so the choice of a programming language could be based on other aspects such as syntactical support for complex structures and extensiveness of software libraries. *ArsecoEngine* has been implemented in C# for those reasons. I emphasized the *extensibility* and *exchangeability* of all components for research purposes.

6.3.2 Composers

Service composition is divided into two logical units: *Service*, which contains the classes related to collaborator description, and *Composer*, which contains the actual composition algorithms. Figure 6.3 below displays the class diagram.

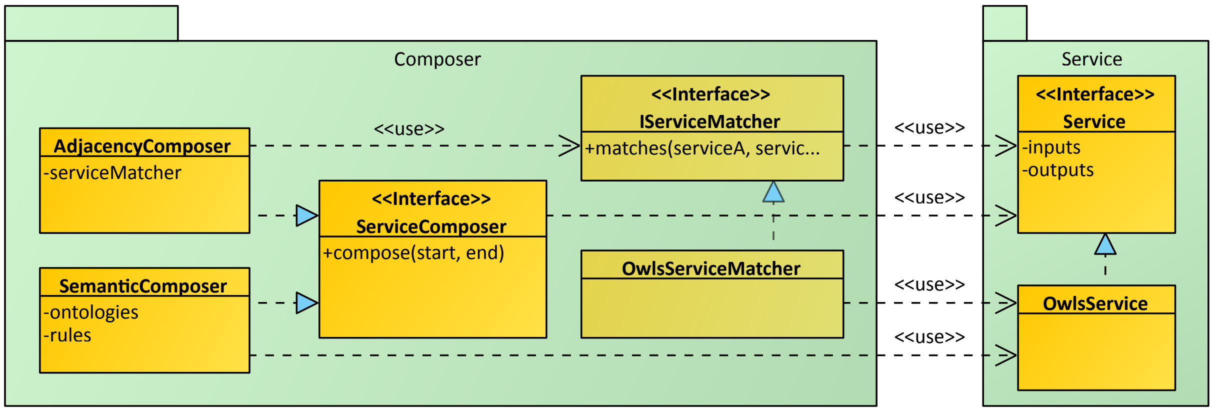


Figure 6.3: Class diagram of the composer components

The `Service` and `ServiceComposer` interfaces are the main entry points. A `Service` represents a generic service that is not bound to any specific description or implementation. A `ServiceComposer` in turn, represents a generic composer that composes services of any type with an unspecified algorithm.

The class diagram clearly indicates the main difference between the different types of composers as outlined in Subsection 4.1.2. The algorithm of `AdjacencyComposer` is independent of a concrete `Service` implementation; the selected `ServiceMatcher` determines which type of services can be composed. By contrast, the `SemanticComposer` directly depends on `OwlsService`, a concrete `Service` implementation for services described by OWL-S.

This separation of concerns demonstrates a high reusability of the components. The use of interfaces will enable us to plug any composer type into the supervisor, which functions independently of the selected composition algorithm.

6.3.3 Supervisor

The supervisor component resides at the top of the component hierarchy. To ensure sufficient modifiability and testability, all dependencies are bound on interfaces rather than concrete classes. The only requirement is that the actual instances are able to communicate using RDF. Below, we present a class diagram of the engine, followed by an overview of the new components involved.

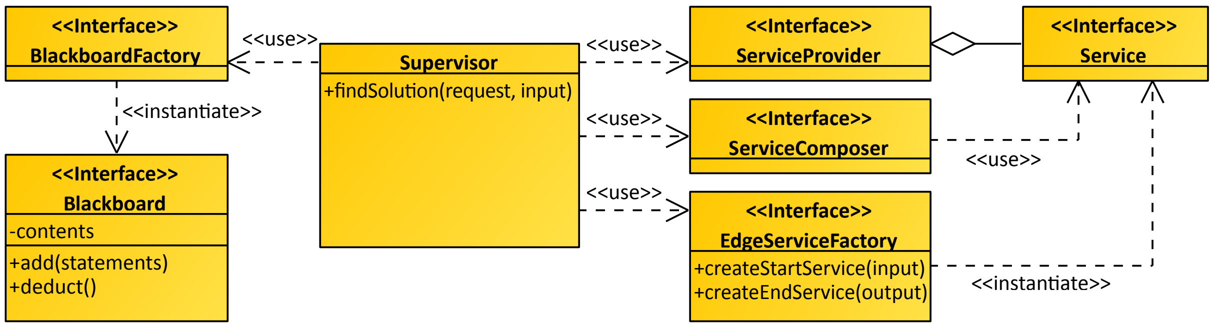


Figure 6.4: Class diagram of the engine

Service provider

A `ServiceProvider` acts as a source of services (collaborators), regardless of their description format. Practical implementations include `XmlServiceProvider`, which loads services from an XML list such as the one generated by the collaborator HTTP server, and `AggregateServiceProvider`, which groups other service providers.

Edge service factory

Composers create compositions from a start service towards an end service, yet the input and request are supplied as RDF statements. An `EdgeServiceFactory` therefore converts these statements into service descriptions understood by the actual `ServiceComposer` instance.

Blackboard factory

We divert slightly from the original specification wherein a blackboard is a simple RDF store. Instead, we provide it with *reasoning capabilities* to deduce new knowledge from existing information, for example by using rules and ontologies. This decision conveniently moves all semantic reasoning capabilities out of the supervisor and places them inside the `Blackboard` and `ServiceComposer` components, each of which can have its own reasoning mechanism. The supervisor thus functions independently of the used reasoning methods. Every request handled by the supervisor requires a new blackboard instance, in contrast to the reusable `ServiceComposer`. A `BlackboardFactory` is the component responsible for the creation of a blackboard with appropriate reasoning instruments.

6.3.4 Semantic Web technologies

The previous section indicated the components in which semantic reasoning takes place. The Semantic Web software libraries used include:

- ***SemWeb*** [42]: an RDF framework for .Net/C# with reasoner support, containing implementations of an RDFS [10] reasoner and the Euler reasoner;
- ***Euler*** [14] and specifically its *Eye* implementation in YAP Prolog: a reasoner that employs ontologies and N3 rules. In addition to the features of the Euler reasoner in SemWeb, Eye supports more built-in functions [4] for use in rules, providing advanced processing capabilities.

Next, we need to decide which ontologies and rules to include in the reasoning process. All ontologies mentioned in any of the collaborators should be preloaded to ensure the recognition of all valid compositions. Additional knowledge, linking different ontologies with similar concepts using RDFS and OWL constructs, will prove convenient as well. Materialization of this knowledge into RDF statements requires semantic rules for RDFS and OWL expressions, such as the N3 rules found in [14]. The example below demonstrates this concept.

Example 6.1 Materializing ontological knowledge into RDF statements

Content: `<Jolly_Jumper> a ex:Horse.`

Ontology: `ex:Horse rdfs:subClassOf ex:Animal.`

To deduce that Jolly Jumper is an animal, we need the following rule that captures the functional meaning of the `subClassOf` predicate (taken from [14]).

Rule: `{?C rdfs:subClassOf ?D. ?X a ?C} => {?X a ?D}.`

Result: `<Jolly_Jumper> a ex:Animal.`

6.3.5 Configuration

As the platform consists of loosely coupled components that interconnect in the supervisor, we need an efficient means of assembling them into a running system suited for the targeted problem domain and the available collaborators and technologies. The architecture has been optimized for a technique called *dependency injection* [18]. At runtime, a *dependency injection framework* instantiates the correct components automatically using an XML-based configuration file. This file references the concrete classes and parameters – such as ontologies and rules – the engine has access to.

The result is a highly modifiable and configurable system that meets the requirements outlined in the beginning of this chapter.

Chapter 7

Use Case

The framework developed so far is a general-purpose semantic problem solver. The employed knowledge and available collaborators determine the problem domain in which a framework instance operates. This chapter discusses a metadata generation use case, illustrating the added value of Semantic Web technologies in metadata problem solving.

7.1 Problem formulation

A publisher of a current affairs magazine has a digital photo archive which needs to be annotated. Apart from the image bitmap data, no additional information is available. As a first step, we would like to identify the people on the photographs, which – given the context of the magazine – will mostly be celebrities.

We dispose of the following collaborators:

- an implementation of the Viola-Jones **face detection** algorithm [46], which finds regions in an image that contain a human face (Subsection 7.2.1);
- an implementation of the **face recognition** algorithm by Verstockt et al. [45], which recognizes a face in a well-delineated region, using a training set (Subsection 7.2.2).

Furthermore, we have access to the following knowledge:

- image, region, and face ontologies and rules;
- Semantic Web knowledge, particularly about celebrities, through *DBpedia [13]*.

For this use case, we direct our attention to the photograph `Loft.jpg`, shown in Figure 7.1. It is a still of the Belgian movie *Loft*, depicting the four main actors. Automated face recognition is hampered by lens blur (actor 1), occlusion with the actor's hand (actor 1), and shadows (actor 3). We investigate how our semantic problem solver handles this image and how it overcomes the aforementioned difficulties.

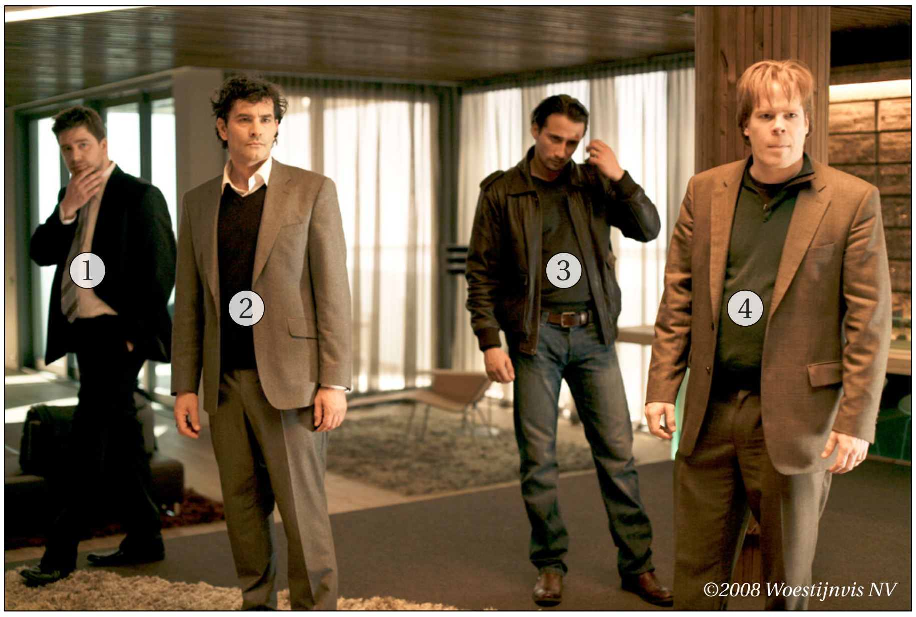


Figure 7.1: Movie still depicting four persons

7.2 Available collaborators

7.2.1 Face detection

Implementation

The face detection collaborator was written in C++, based on the implementation of the Viola-Jones detector in [9, p. 511–512]. It uses the Arseco C++ toolkit and the *OpenCV* computer vision library.

Input and output model

We needed ontologies for images, regions, and faces. The *Friend of a Friend (FOAF)* project contains the `foaf`[1] ontolology, which defines the `Image` class. As part of the *w3photo* project, an image region vocabulary was created under the `imreg`[2] namespace. This vocabulary includes predicates that express the relation between an image and its regions, and between a region and its contents. Finally, we created a `face` ontology and a ruleset complementing the aformentioned ontologies, both listed in Appendix A.

Image regions are indicated by appending a fragment identifier to the URI of the original image, as proposed by the *W3C Media Fragments Working Group* [33]. This technique offers a flexible and format independent approach.

[1]The `foaf` ontology is located at `http://xmlns.com/foaf/0.1/`.
[2]The `imreg` namespace is `http://www.w3.org/2004/02/image-regions#`;
the ontology itself is hosted at `http://www.bnowack.de/w3photo/specs/ontology8`.

OWL-S description

The OWL-S description can be found in Appendix A under Section A.4.

Example interaction

Example 7.1 Face detection request returning two regions

SPARQL query

```
PREFIX : <http://multimedialab.elis.ugent.be/ontologies/arseco/owl-s-sparql.owl#>.
SELECT ?regionList
WHERE {
  [:input [:bindsParameter "image";
           :boundTo <Couple.jpg>];
   :output [:bindsVariable "regionList";
            :boundTo ?regionList]]
}
```

Result (contents of `regionList`)

```
(<People.jpg#xywh=200,100,50,50>
 <People.jpg#xywh=430,80,60,60>)
```

The collaborator replies that *Couple.jpg* contains two faces: one at position $(200,100)$ with size 50×50 and one at position $(430,80)$ with size 60×60.

7.2.2 Face recognition

Implementation

This collaborator is the original algorithm of [45], implemented in MATLAB. It was bundled with an Arseco wrapper to enable the SPARQL protocol.

Input and output model

We use the `imreg` and `face` ontologies for the region input and the face output respectively. The latter ontology also refers to the `foaf` ontology to represent the fact that a face belongs to a person.

OWL-S description

The OWL-S description can be found in Appendix A under Section A.5.

Example interaction

Example 7.2 Face recognition request returning a person

SPARQL query

```
PREFIX : <http://multimedialab.elis.ugent.be/ontologies/arseco/owl-s-sparql.owl#>.
CONSTRUCT { <AlbumCover.jpg> :depicts ?person. }
WHERE {
  [:input [:bindsParameter "region";
           :boundTo <AlbumCover.jpg#xywh=200,30,35,35>];
   :output [:bindsVariable "face";
            :boundTo ?face];
   :output [:bindsVariable "person";
            :boundTo ?person]]
}
```

Result

```
<AlbumCover.jpg> :depicts <Chuck_Mangione>.
```

The collaborator recognizes the person in the image and constructs an RDF graph containing the answer.

7.3 Input and request

The input sent to the problem solver is similar to that below. The `image` parameter is replaced by the actual image URL beforehand.

Example 7.3 Problem solver image input

```
?image a foaf:Image.
```

The request will resemble that of the example below.

Example 7.4 Problem solver person request

```
?image foaf:depicts ?person.
```

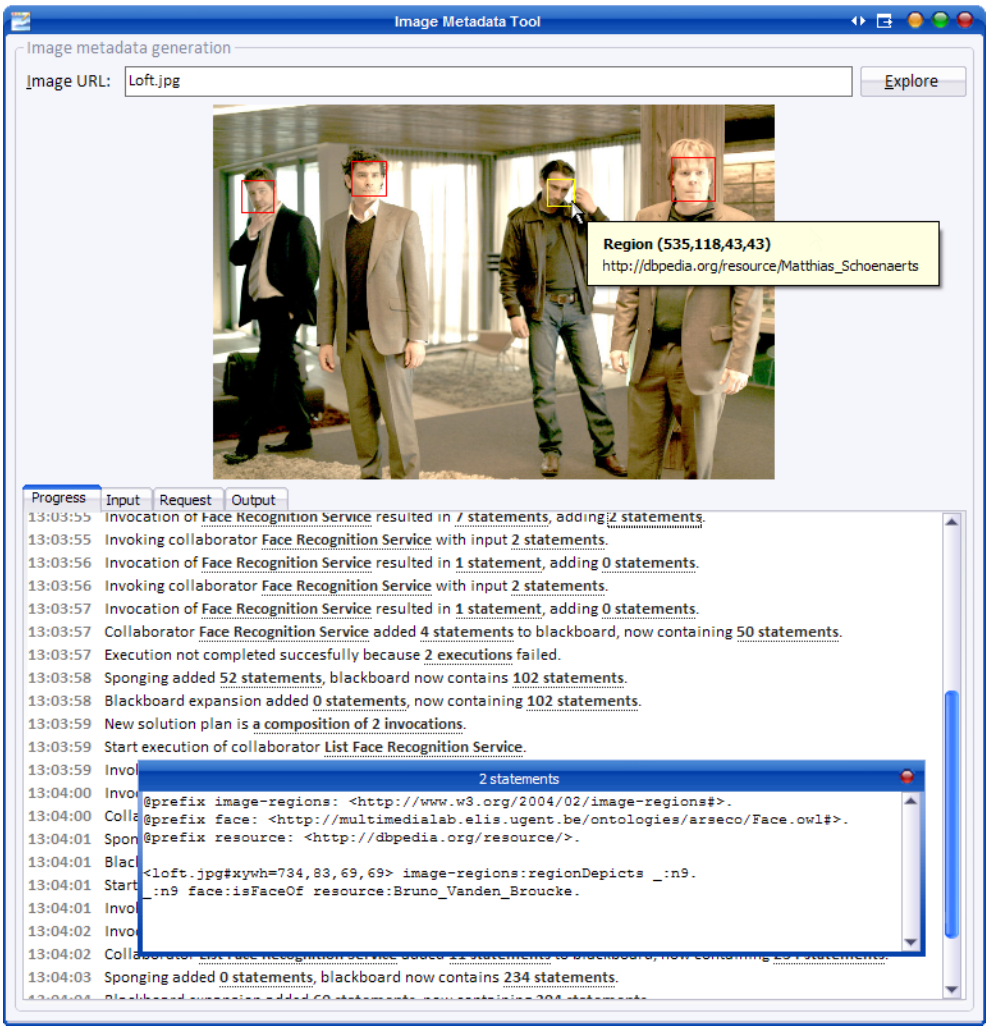
Note how concisely both the input and the request can be formulated. It represents the problem as we humans envision it, in a non-abstract, semantically robust way. The composer will connect the input and request using semantic reasoning, exhibiting an artificial understanding of the problem.

7.4 Supporting software

To support the task of the use case, we created a graphical user interface that specializes in the annotation of images. This program interacts with the supervisor and provides the following facilities:

- **request and input editor**;
- **blackboard and output display**;
- **chronological progress display** with access to details of the current composition, invocations, failures, reasoning results, and past blackboard contents;
- **annotated image display** with the possibility to show details on each annotation.

Supervisor and collaborator settings, as well as the available knowledge and rules can be configured in a file. Although targeted at images, the tool can function as a frontend for any problem request. Below is a screenshot of the program during the solution finding process of this chapter's use case.



Region (535,118,43,43)
http://dbpedia.org/resource/Matthias_Schoenaerts

```
13:03:55  Invocation of Face Recognition Service resulted in 7 statements, adding 2 statements.
13:03:55  Invoking collaborator Face Recognition Service with input 2 statements.
13:03:56  Invocation of Face Recognition Service resulted in 1 statement, adding 0 statements.
13:03:56  Invoking collaborator Face Recognition Service with input 2 statements.
13:03:57  Invocation of Face Recognition Service resulted in 1 statement, adding 0 statements.
13:03:57  Collaborator Face Recognition Service added 4 statements to blackboard, now containing 50 statements.
13:03:57  Execution not completed succesfully because 2 executions failed.
13:03:58  Sponging added 52 statements, blackboard now contains 102 statements.
13:03:58  Blackboard expansion added 0 statements, now containing 102 statements.
13:03:59  New solution plan is a composition of 2 invocations.
13:03:59  Start execution of collaborator List Face Recognition Service.
13:03:59  Invol
13:04:00  Invo
13:04:00  Colla
13:04:01  Spon
13:04:01  Blac
13:04:01  Start
13:04:01  Invol
13:04:02  Invo
13:04:02  Colla
13:04:03  Sponging added 0 statements, blackboard now contains 234 statements.
```

```
@prefix image-regions: <http://www.w3.org/2004/02/image-regions#>.
@prefix face: <http://multimedialab.elis.ugent.be/ontologies/arseco/Face.owl#>.
@prefix resource: <http://dbpedia.org/resource/>.

<loft.jpg#xywh=734,83,69,69> image-regions:regionDepicts _:n9.
_:n9 face:isFaceOf resource:Bruno_Vanden_Broucke.
```

Figure 7.2: Graphical user interface of the image metadata annotation tool

7.5 Execution plan

The supervisor creates a start service from the inputs, and an end service from the request. These are sent to the active composer, which is the reasoner-based composer described in Section 4.5. Although the composing process seems trivial because of the limited number of collaborators and parameters, the realization of this composition – detailed below – will show the contrary. We should also consider the presence of several other collaborators in addition to the ones mentioned, which hinders a prima facie composition.

First, the composer tries to find a path from the request towards the input using backwards reasoning. In this process, it transforms the request by using existing semantic knowledge in the form of a rule, which relates several concepts.

Example 7.5 Semantic knowledge connecting image, person, and region

"An image depicts a person, if it contains a region with the person's face."

```
{
  ?image imreg:region ?region.
  ?region imreg:regionDepicts ?face.
  ?face face:isFaceOf ?person.
}
=>
{ ?image foaf:depicts ?person. }.
```

From this, we can deduce that an image with a region containing the person's face is sufficient to meet the request. However, we do not have a collaborator that connects image, region, face, and person. Clearly, we need a holistic vision on the problem, similar to what we established in Example 4.9. It also indicates why more simple algorithms, such as that of the adjacency composer, cannot solve this request.

The next step is to satisfy each of the three RDF triples in the rule's antecedent. The last two triples correspond to the consequent of the face recognition collaborator rule, generated following the guidelines of Subsection 4.5.3.

Example 7.6 Face recognition collaborator rule

```
{ ?region a face:FaceRegion. }
=>
{
  ?region imreg:regionDepicts ?face.
  ?face a face:Face;
        face:isFaceOf ?person.
  ?person a foaf:Person.
}
```

Now, we are left with two constraints on the region: it should be a part of the image and its type must inherit from `FaceRegion`. If we recall that the `face` ontology (Section A.2) defines `FaceRegion` as a `Region` which depicts a `Face`, it seems obvious that the face detection collaborator rule should fulfill these premises.

Example 7.7 Face detection collaborator rule

```
{ ?image a foaf:Image. }
=>
{
  ?regionList a rdf:List.
  _:region owl:oneOf ?regionList;
           rdf-schema:subClassOf face:FaceRegion,
                                 [owl:hasValue ?image;
                                  owl:onProperty imreg:regionOf].
}
```

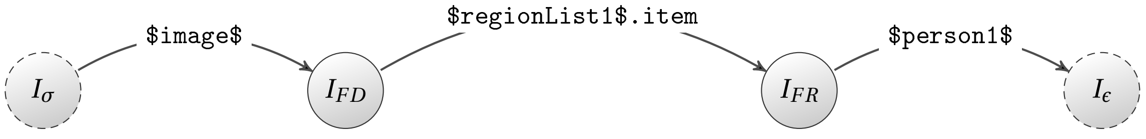
However, there is one slight catch: the face recognition collaborator does not return a face region in the image, but rather a *list* with zero or more of those regions. This reminds us of the discussion in Subsection 4.6.2, where we introduced the concept of *helper services*. A virtual list helper service invocation completes the connection between the face detection and recognition rules, and also acts as a placeholder for the special variable access method.

Finally, a direct link between the face detection rule's antecedent and the input exists. The backwards reasoning process guarantees the connection between the input image and the output person. The composition embodies more than simply the succession of *"finding a face region in an image"* and *"finding a person in a face region"*: it effectively amounts to *"finding **the** person depicted in **the** image"*. Example 7.8 shows the final result.

Example 7.8 Composition from image input to person request

$$\begin{cases} I_\sigma & := & (param1 = \texttt{\$image\$}) \leftarrow \texttt{Input()} \\ I_{FD} & := & (regionList = \texttt{\$regionList1\$}) \leftarrow \texttt{FaceDetection}(image = \texttt{\$image\$}) \\ I_{FR} & := & (face = \texttt{\$face1\$},\ person = \texttt{\$person1\$}) \\ & & \qquad \leftarrow \texttt{FaceRecognition}(region = \texttt{\$regionList1\$.item}) \\ I_\epsilon & := & () \leftarrow \texttt{Request}(person = \texttt{\$person1\$}) \end{cases}$$

I_σ and I_ϵ are the virtual start and end services, respectively.



7.6 Supervision process

We now step through an actual execution of the request with Figure 7.1 as input.

7.6.1 Face detection

The invocation of the face detection collaborator succeeds and returns the following output.

Example 7.9 Face detection collaborator invocation output

```
_:regionList owl-s-sparql:bindsVariable "regionList";
             owl-s-sparql:boundTo (<Loft.jpg#xywh=45,121,51,51>
                                   <Loft.jpg#xywh=221,91,56,56>
                                   <Loft.jpg#xywh=535,118,43,43>
                                   <Loft.jpg#xywh=734,83,69,69>).
<Loft.jpg#xywh=45,121,51,51> a face:FaceRegion;
                             imreg:regionOf <Loft.jpg>.
<Loft.jpg#xywh=221,91,56,56> a face:FaceRegion;
                             imreg:regionOf <Loft.jpg>.
<Loft.jpg#xywh=535,118,43,43> a face:FaceRegion;
                              imreg:regionOf <Loft.jpg>.
<Loft.jpg#xywh=734,83,69,69> a face:FaceRegion;
                             imreg:regionOf <Loft.jpg>.
```

The four resource URIs are assigned to the `$regionList1$` variable, as instructed by the composition. Visual inspection of this output reveals that the detector finds the faces correctly, as depicted in Figure 7.3 below.

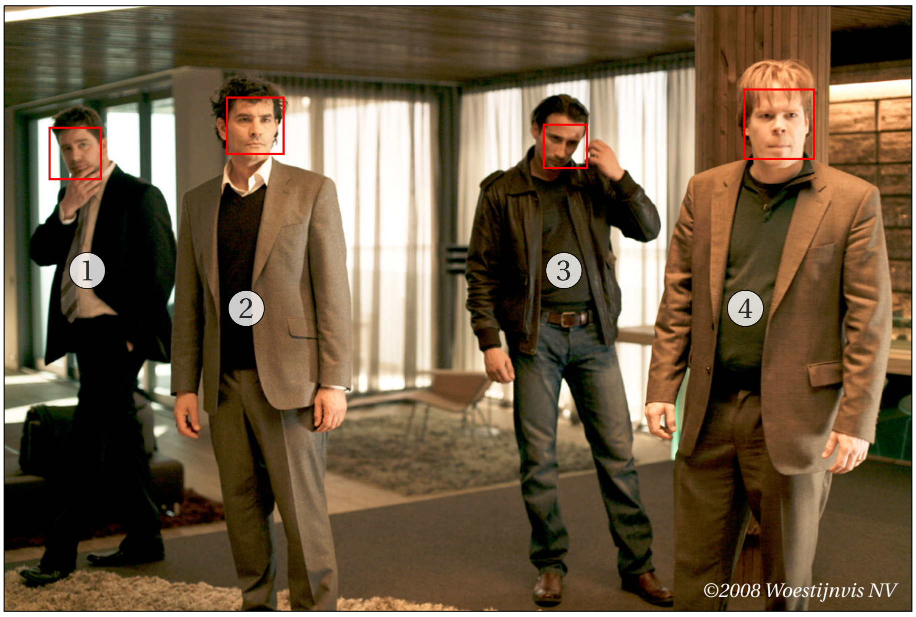


Figure 7.3: Face detection results highlighted in the original image

7.6.2 Face recognition

We now proceed with the face recognition collaborator invocation. The composition demands that this is executed for every item assigned to the `$regionList1$` variable, which in our case amounts to four invocations.

Example 7.10 Face recognition collaborator invocation output

Output with *region* = `<Loft.jpg#xywh=45,121,51,51>` (actor 1)

(none)

Output with *region* = `<Loft.jpg#xywh=221,91,56,56>` (actor 2)

```
@prefix dbpedia: <http://dbpedia.org/resource/>.
_:face owl-s-sparql:bindsVariable "face";
       owl-s-sparql:boundTo [face:isFaceOf dbpedia:Koen_De_Bouw].
_:person owl-s-sparql:bindsVariable "person";
         owl-s-sparql:boundTo dbpedia:Koen_De_Bouw.
```

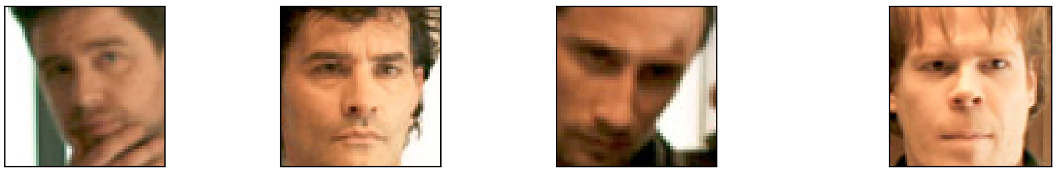
Output with *region* = `<Loft.jpg#xywh=535,118,43,43>` (actor 3)

(none)

Output with *region* = `<Loft.jpg#xywh=734,83,69,69>` (actor 4)

```
_:face owl-s-sparql:bindsVariable "face";
       owl-s-sparql:boundTo [face:isFaceOf dbpedia:Bruno_Vanden_Broucke].
_:person owl-s-sparql:bindsVariable "person";
         owl-s-sparql:boundTo dbpedia:Bruno_Vanden_Broucke.
```

It is apparent that this part of the execution did not pass off as smoothly as the previous step. We were able to find two of the four assignments for `$person1$`, but the others failed. Looking at Figure 7.4, we can understand why the algorithm failed. The correctly recognized faces of actors 2 and 4 were relatively easy to detect because of their orientation, illumination and contrast. The face of actor 1 is harder to recognize because it appears slightly out of focus and the actor's right hand rests on his chin. The left hand of actor 3 casts a shadow on his face which decreases the image contrast locally, interfering with feature extraction.



(a) *unrecognized* (b) `dbpedia:Koen_De_Bouw` (c) *unrecognized* (d) `dbpedia:Bruno_Vanden_Broucke`

Figure 7.4: Face recognition results on individual regions

7.6.3 Failure recovery

Failure detection and impact determination

Two face detection invocations do not return an answer, and the supervisor classifies this as a failure. The impacted part spans the face detection invocation and the end service. Peculiarly, this impact is only partial in that half of the needed values are available. Consequently, the adaptation only needs to find alternatives for the two failed invocations.

Blackboard enrichment

Prior to the generation of a new plan, the supervisor tries to enrich the blackboard by deriving new semantic knowledge. This enrichment is a combination of semantic inference and a technique known as *sponging*, done quite commonly in a Semantic Web context. To sponge means to look up related information using semantic data sources. Spongers can be seen as a specialized kind of collaborators, in that they are not involved directly in the execution plan, but called whenever additional information is required.

In this case, we dispose of a collaborator that accesses DBpedia. Typical queries will be similar to the following.

Example 7.11 DBPedia sponging queries for Koen De Bouw

Query selecting triples with the entity as subject

```
PREFIX dbpedia: <http://dbpedia.org/resource/>
CONSTRUCT { dbpedia:Koen_De_Bouw ?p ?o. }
WHERE {
  dbpedia:Koen_De_Bouw ?p ?o.
}
```

Query selecting triples with the entity as object

```
PREFIX dbpedia: <http://dbpedia.org/resource/>
CONSTRUCT { ?s ?p dbpedia:Koen_De_Bouw. }
WHERE {
  ?s ?p dbpedia:Koen_De_Bouw.
}
```

These queries are to be executed for all entities on the blackboard, possibly in a limited recursive way. In practice, however, the sponger can limit its search to entities whose URI starts with the `dbpedia` prefix, since those are the only entities present on DBpedia. On the current blackboard, the identifiers of actor 2 and actor 4 qualify this criterion.

Upon retrieval of our queries, DBpedia accesses its extensive database and returns knowledge which is then combined, as displayed below.

Example 7.12 Related knowledge retrieved from DBpedia *(truncated)*

```
@prefix dbpedia: <http://dbpedia.org/resource/>.
@prefix dbpprop: <http://dbpedia.org/property/>.
@prefix dbpedia-owl: <http://dbpedia.org/ontology/>.
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#>

dbpedia:Koen_De_Bouw a dbpedia-owl:Actor;
                     rdfs:label "Koen De Bouw".

dbpedia:Bruno_Vanden_Broucke a dbpedia-owl:Actor;
                             rdfs:label "Bruno Vanden Broucke".

dbpedia:Lijmen%2FHet_Been dbpprop:director dbpedia:Robbe_De_Hert;
                          dbpprop:name "Lijmen/Het Been";
                          dbpprop:starring dbpedia:Koen_De_Bouw,
                                           dbpedia:Mike_Verdrengh,
                                           dbpedia:Sylvia_Kristel,
                                           dbpedia:Willeke_van_Ammelrooy;
                          dbpedia-owl:writer dbpedia:Willem_Elsschot,
                                             dbpedia:Robbe_De_Hert,
                                             dbpedia:Fernand_Auwera.

dbpedia:Loft_%282008_film%29 dbpprop:director dbpedia:Erik_Van_Looy;
                             dbpprop:name "Loft";
                             dbpprop:producer dbpedia:Woestijnvis;
                             dbpprop:starring dbpedia:Bruno_Vanden_Broucke,
                                              dbpedia:Filip_Peeters,
                                              dbpedia:Jan_Decleir,
                                              dbpedia:Koen_De_Bouw,
                                              dbpedia:Koen_De_Graeve,
                                              dbpedia:Matthias_Schoenaerts,
                                              dbpedia:Veerle_Baetens.
```

The returned information semantically relates to the entities `dbpedia:Koen_De_Bouw` and `dbpedia:Bruno_Vanden_Broucke`. Since they both refer to actors, DBpedia contains details about the movies they starred in. Some other facts were omitted here for brevity, since they are not used in the remainder of this section.

The amount of information returned is enormous in comparison to the limited input of only two entities, yet this poses no problem. Firstly, the reasoner will automatically use the triples which it can relate to other statements or rules. Secondly, the additional amount of information can prove interesting for the output.

Plan adaptation

The question now is how this additional information can help us in repairing the composition. The relationship between the people in the photograph can assist us, since they often relate to each other in some way. Inversely, we can safely assume that the statistical probability to appear in the same photograph is significantly higher with acquaintances compared to random people. This fact inspired the following N3 rule.

Example 7.13 Rule that suggests the depiction of acquaintances in an image

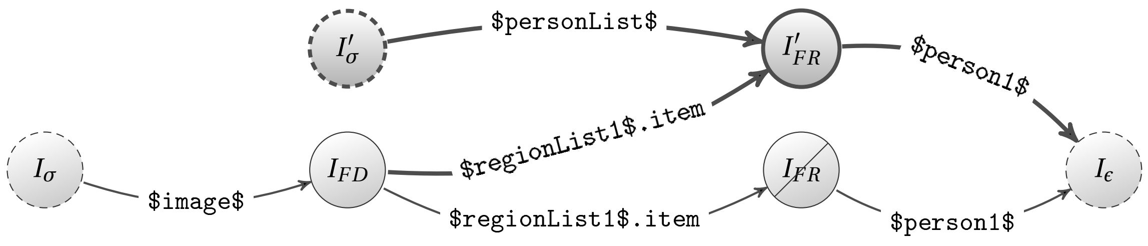
```
{
  ?image foaf:depicts ?person
  ?person foaf:knows ?acquaintance.
}
=>
{ ?image face:maybeDepicts ?acquaintance. }.
```

Furthermore, we can also assume that people know each other if a working relationship exists between them. If two actors co-star in a movie, this implies a working relationship. We can also capture these facts in simple N3 rules. In combination with the derived DBpedia knowledge, we find several people acquainted with the detected actors.

The collaborator can now employ this knowledge to guide the face recognition collaborator. The latter has an optional `candidates` parameter, by which we can suggest faces for the recognition process. In response, the collaborator can temporarily boost the probability of those faces in its internal training set, enabling a more pronounced recognition result. The supervisor adapts the composition by adding a new invocation, adding the derived acquaintances to the `candidates` parameter. Note that the actual process is slightly more complex, but some details were omitted for brevity.

Example 7.14 Adapted composition, recovering from the face recognition failure

$$\left\{\begin{array}{lll} \vdots & & \\ I'_\sigma & := & (param1 = \texttt{\$personList\$}) \leftarrow \texttt{Input()} \\ I'_{FR} & := & (face = \texttt{\$face1\$},\ person = \texttt{\$person1\$}) \\ & & \qquad\qquad \leftarrow \texttt{FaceRecognition}(region = \texttt{\$regionList1\$.item}, \\ & & \qquad\qquad\qquad\qquad candidates = \texttt{\$personList\$}) \end{array}\right.$$



7.6.4 Face recognition (bis)

The execution of the second face recognition invocation returns the following values.

Example 7.15 Second face recognition collaborator invocation output

Output with *region* = `<Loft.jpg#xywh=535,118,43,43>` (actor 3)

```
_:face owl-s-sparql:bindsVariable "face";
       owl-s-sparql:boundTo [face:isFaceOf dbpedia:Matthias_Schoenaerts].
_:person owl-s-sparql:bindsVariable "person";
         owl-s-sparql:boundTo dbpedia:Matthias_Schoenaerts.
```

Output with *region* = `<Loft.jpg#xywh=45,121,51,51>` (actor 1)

```
_:face owl-s-sparql:bindsVariable "face";
       owl-s-sparql:boundTo [face:isFaceOf ?person].
_:person owl-s-sparql:bindsVariable "person";
         owl-s-sparql:boundTo ?person.
({?person = dbpedia:Koen_De_Graeve.}
 {?person = dbpedia:Bruno_Vanden_Broucke.}) e:disjunction [a e:T].
```

The additional information has proven useful: actor 3 is recognized correctly as the entity `dbpedia:Matthias_Schoenaerts`. However, the algorithm still doubts between two alternatives for actor 1 and returns both options, indicating its uncertainty. Semantic knowledges again comes to the rescue, in the form of the following rule.

Example 7.16 A photograph cannot depict the same person twice

```
{
  ?image imreg:region ?regionA, ?regionB.
  ?regionA owl:differentFrom ?regionB.
  ?regionA imreg:regionDepicts [face:isFaceOf ?personA].
  ?regionB imreg:regionDepicts [face:isFaceOf ?personB].
}
=>
{ ?personA owl:differentFrom ?personB. }.
```

This rule executes for every detected face in the image, resulting in three facts for actor 1: it cannot be the same person as actor 2, nor actor 3, nor actor 4. Using the predicate logic rule of the neutral element for disjunction $\{p \vee q, \neg q\} \vdash p$ and the fact that actor 4 was recognized as `dbpedia:Bruno_Vanden_Broucke`, the supervisor deduces that actor 1 can only be `dbpedia:Koen_De_Graeve`. Both results are assigned to `$person1$`, which now totals four values, finishing the execution of the composition.

7.7 Response output

Finally, the rule of Example 7.5 executes, this time in the forward direction. This links the actors to the original image, completing the demand. Thanks to the successful error recovery, the supervisor solved the request in its entirety. We summarize the response output below.

Example 7.17 Response output of the image request

```
<Loft.jpg> foaf:depicts dbpedia:Bruno_Vanden_Broucke,
                        dbpedia:Koen_De_Bouw,
                        dbpedia:Koen_De_Graeve,
                        dbpedia:Matthias_Schoenaerts.
```

7.8 Concluding remarks

Semantic Web technologies played a crucial role in the use case of this chapter. We list three advantages over traditional metadata generation tools:

- **semantic input and request**, which enables us to state the metadata generation problem in a formal, yet human-understandable way;
- **intelligent composition**, which generates a solution method adapted to a specific input and request;
- **advanced problem resolution**, which uses Semantic Web knowledge to assist feature extraction algorithms.

This use case also clearly illustrates the concept of *semantic feedback* (Subsection 2.3.1) and the bidirectional cooperation between the collaborators and the solution. Feature extraction algorithms contribute to the solution and the solution in turn improves the operation of these algorithms. This concept enables more powerful metadata extraction capabilities.

The software, files, and a video demonstration of this chapter can be found on the accompanying DVD (see Appendix C).

Chapter 8

Future Opportunities

This chapter presents several challenges in the field of semantic problem solving, as a starting point for future research. Topics include quality metrics, the handling of imperfect information, knowledge modeling, the application to different problem domains, and the incorporation of user interaction.

8.1 Quality metrics

The use case of the platform proposed in this dissertation, is merely a proof-of-concept. In order to direct our semantic problem solving research towards practical applications, we need to qualify parameters that measure its achievements. Specifically, we must be able to quantify the added value that Semantic Web technologies provide.

One approach is to define a set of standardized tests that cover common tasks. The solutions of different problem solvers should be compared using relevant metrics such as:

- **correctness:** the extent to which the returned results are accurate and, if applicable, relevant to the original request *(cf. "precision" of search engines)*;
- **completeness:** the amount of information found compared to the requested amount, possibly considering additional information generated as a side-effect *(cf. "recall")*;
- **performance:** how many resources (time, memory, budget, etc.) were consumed.

These should be measured under varying circumstances such as different collaborators and knowledge sources. Fair comparisons are often difficult, since the contribution of human customizations is hard to measure. Traditional algorithms on the one hand, require a lot of parameter tweaking. The semantic platform, on the other hand, requires sufficient knowledge to adapt the parameters autonomously.

Specifically for metadata extraction problems, we should study current practical uses and investigate how to integrate a semantic problem solver in the workflow. Subsequently, this solution must be evaluated using the above metrics.

8.2 Imperfect information

8.2.1 Impact of imperfections

Metadata extraction algorithms often return results that are associated with some kind of probability. For example, in the case of object recognition, the result could be the best match with a certain training set. It is not necessarily *accurate* and *certain*: the match could be a false positive, and even if the match is correct, the algorithm does not know that for sure.

Clearly, a real-world problem solving system must be able to deal with imperfections. We distinguish several orthogonal dimensions of imperfection [34]:

- **inaccuracy**: the accuracy is less than expected or required;
  Examples: *a $2 \cdot 10^2$ cm tall person, a color between yellow and red*
- **vagueness**: the description is inherently vague;
  Examples: *a tall person, a bright color*
- **uncertainty**: the degree of truth can vary from false to true;
  Examples: *the person is possibly tall, we are 50% certain the color is orange*
- **incompleteness**: parts of the information are missing;
  Examples: *a person with an unknown name, a color with unspecified saturation*
- **inconsistency**: the information contradicts itself.
  Examples: *a person with two birth dates, a dark color between yellow and white*

The platform currently deals well with incompleteness, which is made up for by semantic knowledge and execution plan adaptation. A limited form of uncertainty, namely the case of multiple alternatives for a parameter, is also handled. Vagueness and inaccuracy can also be considered to some extent by including appropriate N3 rules.

However, since the reasoner employs a monotonic reasoning process, there is no way to deal with inconsistency. The *principle of explosion* or *ex falso quodlibet* holds: $\{p, \neg p\} \vdash q$. This means that, if one algorithm recognizes an object as A and another recognizes it as B, *any* statement can be deduced, rendering all found information essentially worthless. Fortunately, we can often trade inconsistency for uncertainty: while the statements *"the object is A"* and *"the object is B"* cannot coexist, the statements *"the object **could** be A"* and *"the object **could** be B"* can.

While the other dimensions are somewhat supported, a solid mechanism for handling imperfections is indispensable to offer usable results. Currently, the supervisor considers all information perfect, silently propagating imperfections into the output as uncertainties. The output is represented as a truth, but how trustworthy is this if the intermediate values were results of probabilistic algorithms? The fact that the final degree of uncertainty currently remains unknown to the requester, does not improve the situation.

8.2.2 Possible approaches

The *W3C Uncertainty Reasoning for the World Wide Web Incubator Group* created a report on handling imperfect information [29]. Two approaches are pushed forward: probability theory (for example, using Bayesian Networks) and fuzzy logic. The report also provides an ontology for different types of imperfections and use cases that include reasoning.

For concrete applications, an extension to the N3 logic framework exists in the `log-rules` or `e` namespace[1], which we used earlier to represent disjunctions. Revising the use case of Chapter 7 to incorporate imperfections, we could model the rule of Example 7.13 as follows.

> **Example 8.1** Rule that suggests the depiction of acquaintances in an image
>
> ```
> {
>   ?image foaf:depicts ?person.
>   ?person foaf:knows ?acquaintance.
> }
> =>
> {
>   ({?image foaf:depicts ?acquaintance.}) e:conditional 0.2.
> }.
> ```
>
> The above rule expresses that, if an image depicts a person, there is a 20% chance it also depicts an acquaintance of that person. Here, this number is an ungrounded guess, but in practice, it could be based on statistical observations.

Similarly, collaborators could attach a rating to their output, expressed as a probability representing uncertainty or inaccuracy. This also has its repercussions on the result.

> **Example 8.2** Response output of the image request with fictitious certainty ratings
>
> ```
> ({<Loft.jpg> foaf:depicts dbpedia:Bruno_Vanden_Broucke}) e:conditional 0.83.
> ({<Loft.jpg> foaf:depicts dbpedia:Koen_De_Bouw}) e:conditional 0.76.
> ({<Loft.jpg> foaf:depicts dbpedia:Koen_De_Graeve}) e:conditional 0.61.
> ({<Loft.jpg> foaf:depicts dbpedia:Matthias_Schoenaerts}) e:conditional 0.68.
> ```

Interestingly, the certainty about the result can be included as a *quality metric* when comparing to other problem solving platforms. Furthermore, Semantic Web knowledge could reinforce our belief about the validity of certain results. This would shift the focus of accuracy assessment from individual algorithms, indicating the reliability of their own results, to the problem solver as a whole, determining the certainty about the entire solution.

[1]The e namespace is located at `http://eulersharp.sourceforge.net/2003/03swap/log-rules`.

8.3 Knowledge modeling

As mentioned earlier, the main obstacle for semantic reasoning is the limited availability of relevant semantic knowledge. For some application domains, the amount of RDF linked data and associated OWL ontologies is sufficient.

While ontologies enable powerful relationships such as inheritance, transcendence, and opposition, they do not cover all kinds of possible descriptive knowledge in the human mind. General n-ary relationships with $n \geq 3$ cannot be represented natively, yet they are quite common. For instance, in Example 7.16, we wanted to state that an image cannot contain the same person twice. To express these n-ary relationships, we currently need a logic rule system such as N3 rules. Unfortunately, the public presence of logic rules in any application domain is virtually non-existent. In practice, it is limited to general rules on ontologies that materialize the semantics of the ontological relationships.

Either we should investigate more powerful expression mechanisms to phrase these knowledge elements in a more descriptive style, as opposed to the logic style inherent to rules, or either we should facilitate the creation of such rules.

In favor of the first method, we could argue that the human mind does not approach statements such as *"a photograph can depict persons"* differently from statements such as *"a photograph depicts different persons"*. They both handle about a photograph and persons, yet the former can be expressed in OWL, while the latter necessitates an N3 rule. Many different logic rules can represent the same statement – as can RDF triples – but their application ranges can differ and reasoning about their equality is harder. For example, the rule *"the depiction of two persons in an image implies they are different beings"* conveys the exact same information as *"two depictions of the same person imply that these depictions are in different images"*. However, only the first rule applies to the use case of Chapter 7.

On the other hand, we should not risk complicating ontology creation. An implementation of the second method could consist of an automatic rule generating system. One such approach is offered by the data mining technique of association rules [1], which detects statistical co-occurrences in item sets. For example, if we dispose of 1.000 annotated photographs, of which 200 depict acquaintances, an association rule generating algorithm can derive an association with 20% support. This would result in the rule of Example 8.1.

Complementary to the problem of knowledge availability, it is difficult to estimate the amount of knowledge necessary to solve requests. In the current platform version, the supervisor executes all possible deductions after each invocation. This could result in an excessive amount of superfluous information, which could severely hinder performance and influence the relevance of the returned results negatively. Ideally, the task of humans should be to supply pointers to as many knowledge sources as possible, from which the problem solver should distill an applicable selection.

8.4 Different application domains

To port the concept to domains different from image annotation, we need to tackle several issues. For example, when considering the annotation of – perhaps real-time – video, typically more invocations of more collaborators are necessary.

The challenges here are to provide a comprehensive display of the process and to parallelize the execution of the declarative execution plan. As always, the selection and modeling of appropriate knowledge and collaborators are essential.

8.5 User interaction

The problem solving platform ultimately interacts with an end user. We distinguish several forms of interaction:

- **input and request communication**: the user formulates a problem to which the platform generates an answer, as discussed previously;
- **assistance**: the user is an active agent in one or several collaborator tasks;
- **feedback**: the user inspects the proposed solution and assesses its accuracy.

Human assistance has already been suggested in Subsection 2.3.2, since several tasks are currently performed better by humans. To make this work, we need to create a graphical interface which displays an appropriate representation of the request. The user must be able to communicate in a human-centered model, which is translated to the internal RDF format afterwards. The items under annotation, typically in large batches, need to integrate into the user's workflow. For each item, the supervisor creates a composition, whose execution is scheduled such that human-assisted tasks are grouped together. These can subsequently be performed when qualified personnel is available.

Finally, the platform can learn from past experiences if a user inspects the accuracy of previously generated solutions. Section 8.3 already hinted at the use of history to derive association rules. We could take this idea further and let the selection of certain collaborators depend on how they functioned on similar problems in the past. This statistical information can significantly improve future results.

If the results are biased or if their degree of satisfaction is subjective, the statistical information can be associated with a particular user, effectively creating a profile. For example, if one employee mainly annotates photographs of sport events, the platform adapts its face detection algorithm by favoring athletes.

It is clear that real-world applications will require well-founded user interaction mechanisms, since the domain of metadata generation concerns human-centered solutions.

Chapter 9

Conclusion

In this dissertation, we investigated how to apply Semantic Web technologies to the metadata generation process for multimedia data. We developed a generic software platform which implements the outlined theoretical concepts and indicated future research paths.

First, we presented the architecture of a general problem solver, structured as a blackboard system. A supervisor coordinates several independent collaborators, each of which solves a specific subtask. The blackboard serves as a central place that shares the collected information in the RDF format.

Collaborators are regular algorithms functioning as SPARQL endpoints, solving tasks such as feature extraction or logic inferencing. Existing algorithms can be adapted to this protocol; new algorithms can be created directly as SPARQL processors. They are accompanied by an OWL-S description, detailing their requirements and capabilities.

Individual collaborators are combined by a composer into an execution plan, able to solve complex requests given a certain input. A composer algorithm based on the adjacency matrix of the collaborators can create elementary chain compositions. Reasoner-based composers establish more advanced compositions using Semantic Web knowledge, maintaining a holistic vision on the request.

A supervisor interprets the composition as a declarative program and transforms it into an actual execution. It offers a comprehensive display of the progress of the problem solving process. Collaborator failures result in an adaptation of the original plan, aided by the semantic contents of intermediary results. At the end of the execution, the supervisor formulates an answer based on the blackboard information.

The semantic problem solver was implemented as a platform dubbed *Arseco*, which also provides facilities to develop collaborators. Special care has been taken to provide maximum modifiability, interoperability, and reusability, so that the platform can be adapted to new research domains and applications.

A use case, annotation of an image with person names, demonstrated the adoption of the platform for metadata generation problems. It indicated the added value of Semantic Web technologies by deriving metadata which traditional algorithms cannot find autonomously.

Finally, we identified the current limitations of the platform, pointing out opportunities for future research. Practical applications require the definition of quality metrics for evaluation and comparison, as well as a mechanism for handling imperfect information. We need to examine ways to manage knowledge sources and to extend the application domain to other problems. As a last step, an interaction scheme for end users should be designed.

Appendix A

Ontologies, Vocabularies, and Rules

A.1 `owl-s-sparql` ontology

```
@prefix : <http://multimedialab.elis.ugent.be/ontologies/arseco/owl-s-sparql.owl#>.
@prefix dc: <http://purl.org/dc/elements/1.1/>.
@prefix Expression: <http://www.daml.org/services/owl-s/1.1/generic/Expression.owl#>.
@prefix owl: <http://www.w3.org/2002/07/owl#>.
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#>.
@prefix Service: <http://www.daml.org/services/owl-s/1.1/Service.owl#>.
@prefix sd: <http://www.w3.org/2009/sparql/service-description#>.
@prefix xsd: <http://www.w3.org/2001/XMLSchema#>.

:N3-Expression a owl:Class;
               rdfs:subClassOf Expression:Expression,
                               [a owl:Restriction;
                                owl:onProperty Expression:expressionBody;
                                owl:allValuesFrom xsd:string],
                               [a owl:Restriction;
                                owl:onProperty Expression:expressionLanguage;
                                owl:hasValue :N3].

:N3-Condition a owl:Class;
              rdfs:subClassOf :N3-Expression,
                              Expression:Condition;
              dc:description "An condition in Notation3-format.".

:SparqlQueryForm a owl:Class;
                 owl:equivalentClass [a owl:Class;
                                      owl:oneOf (:SparqlQueryFormSelect
                                                 :SparqlQueryFormDescribe
                                                 :SparqlQueryFormConstruct
                                                 :SparqlQueryFormAsk)].
```

```
:SparqlService a owl:Class;
               owl:equivalentClass [a owl:Class;
                                    owl:intersectionOf
                                      (Service:Service
                                       [a owl:Restriction;
                                        owl:onProperty Service:supports;
                                        owl:someValuesFrom :SparqlServiceGrounding])].

:SparqlServiceGrounding a owl:Class;
                        owl:equivalentClass sd:Endpoint;
                        rdfs:subClassOf Service:ServiceGrounding.

:SparqlVersion a owl:Class;
               owl:equivalentClass [a owl:Class;
                                    owl:oneOf (:SparqlVersionUpdate1_0
                                               :SparqlVersionQuery1_1
                                               :SparqlVersionQuery1_0)].

:supportsQueryForm a owl:ObjectProperty;
                   rdfs:range :SparqlQueryForm;
                   rdfs:domain :SparqlServiceGrounding.

:supportsSparqlVersion a owl:ObjectProperty;
                       rdfs:domain :SparqlServiceGrounding;
                       rdfs:range :SparqlVersion.

:N3 a Expression:LogicLanguage.

:SparqlQueryFormAsk rdfs:label "ASK"^^xsd:string.
:SparqlQueryFormConstruct rdfs:label "CONSTRUCT"^^xsd:string.
:SparqlQueryFormDescribe rdfs:label "DESCRIBE"^^xsd:string.
:SparqlQueryFormSelect rdfs:label "SELECT"^^xsd:string.

:SparqlVersionQuery1_0 rdfs:label "SPARQL/Query 1.0"^^xsd:string.
:SparqlVersionQuery1_1 rdfs:label "SPARQL/Query 1.1"^^xsd:string.
:SparqlVersionUpdate1_0 rdfs:label "SPARQL/Update 1.0"^^xsd:string.
```

A.2 `face` ontology

```
@prefix : <http://multimedialab.elis.ugent.be/ontologies/arseco/Face.owl#>.
@prefix dc: <http://purl.org/dc/elements/1.1/>.
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#>.
@prefix imreg: <http://www.w3.org/2004/02/image-regions#>.
@prefix foaf: <http://xmlns.com/foaf/0.1/>.
@prefix owl: <http://www.w3.org/2002/07/owl#>.

:Face a owl:Class;
      dc:description "A human face.".

:FaceRegion a owl:Class;
            owl:equivalentClass [a owl:Class;
                                 owl:intersectionOf
                                 (imreg:Region
                                  [a owl:Restriction;
                                   owl:onProperty imreg:regionDepicts;
                                   owl:someValuesFrom :Face]
                                 )];
            dc:description "Region that depicts a human face.".

:hasFace a owl:ObjectProperty;
         dc:description "Indicates the face of the person.";
         rdfs:range :Face;
         rdfs:domain foaf:Person;
         owl:inverseOf :isFaceOf.

:isFaceOf a owl:ObjectProperty;
          dc:description "Indicates the person the face belongs to.".
```

A.3 Rules

A.3.1 Person, face and image rules

```
@prefix face: <http://multimedialab.elis.ugent.be/ontologies/arseco/Face.owl#>.
@prefix foaf: <http://xmlns.com/foaf/0.1/>.
@prefix imreg: <http://www.w3.org/2004/02/image-regions#>.
@prefix owl: <http://www.w3.org/2002/07/owl#>.

# An image that contains a region with the face of a person, depicts that person.
{
  ?image imreg:region ?region.
  ?region imreg:regionDepicts ?face.
  ?face face:isFaceOf ?person.
}
=>
{ ?image foaf:depicts ?person. }.

# If an image depicts two different faces, they belong to two different persons.
{
  ?image imreg:region ?regionA, ?regionB.
  ?regionA owl:differentFrom ?regionB.
  ?regionA imreg:regionDepicts [face:isFaceOf ?personA].
  ?regionB imreg:regionDepicts [face:isFaceOf ?personB].
}
=>
{ ?personA owl:differentFrom ?personB. }.

# An image that depicts a person, maybe depicts an acquaintance of his.
{
  ?image foaf:depicts ?person.
  ?person foaf:knows ?acquaintance.
}
=>
{ ?image face:maybeDepicts ?acquaintance. }.
```

A.3.2 DBpedia rules

```
@prefix dbpprop: <http://dbpedia.org/property/>.

# Two actors that star in the same movie, know each other.
{
  ?movie dbpprop:starring ?actor1, ?actor2.
  ?actor1 owl:differentFrom ?actor2.
}
=>
{ ?actor1 foaf:knows ?actor2. }.
```

A.4 Face detection collaborator OWL-S description

```
@prefix : <http://multimedialab.elis.ugent.be/ontologies/
                                              arseco/FaceDetectionService.owl#>.
@prefix dc: <http://purl.org/dc/elements/1.1/>.
@prefix Service: <http://www.daml.org/services/owl-s/1.1/Service.owl#>.
@prefix sd: <http://www.w3.org/2009/sparql/service-description#>.
@prefix Profile: <http://www.daml.org/services/owl-s/1.1/Profile.owl#>.
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#>.
@prefix owl2xml: <http://www.w3.org/2006/12/owl2-xml#>.
@prefix Process: <http://www.daml.org/services/owl-s/1.1/Process.owl#>.
@prefix xsd: <http://www.w3.org/2001/XMLSchema#>.
@prefix owl: <http://www.w3.org/2002/07/owl#>.
@prefix owl-s-sparql: <http://multimedialab.elis.ugent.be/ontologies/
                                                          arseco/owl-s-sparql.owl#>.
@prefix Expression: <http://www.daml.org/services/owl-s/1.1/generic/Expression.owl#>.

:FaceDetectionService a Service:Service;
                      Service:presents :FaceDetectionProfile;
                      Service:describedBy :FaceDetectionProcess;
                      Service:supports :FaceDetectionSparqlGrounding.

:FaceDetectionProfile a Profile:Profile;
                      Profile:serviceName "Face Detection Collaborator";
                      Profile:textDescription "Service that detects face regions
                                                                inside an image.";
                      Profile:has_process :FaceDetectionProcess;
                      Profile:hasInput :Image;
                      Profile:hasOutput :RegionList.

:FaceDetectionProcess a Process:AtomicProcess;
                      Process:hasInput :Image;
                      Process:hasOutput :RegionList;
                      Process:hasResult :FaceDetectionResult.

:FaceDetectionSparqlGrounding a owl-s-sparql:SparqlServiceGrounding;
           owl-s-sparql:supportsQueryForm owl-s-sparql:SparqlQueryFormConstruct,
                                          owl-s-sparql:SparqlQueryFormSelect;
           owl-s-sparql:supportsSparqlVersion owl-s-sparql:SparqlVersionQuery1_0;
           sd:url <http://multimedialab.elis.ugent.be/services/FaceDetection/sparql>.

:Image Process:Input;
       Process:parameterType "http://xmlns.com/foaf/0.1/Image"^^xsd:anyURI.

:RegionList a Process:Output;
  Process:parameterType "http://www.w3.org/1999/02/22-rdf-syntax-ns#List"^^xsd:anyURI.
```

```
:FaceDetectionResult a Process:Result;
                     Process:hasEffect :FaceDetectionEffect.

:FaceDetectionEffect a owl-s-sparql:N3-Expression;
                     Expression:expressionBody
  """@prefix owl: <http://www.w3.org/2002/07/owl#>.
     @prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#>.
     @prefix imreg: <http://www.w3.org/2004/02/image-regions#>.
     @prefix face: <http://multimedialab.elis.ugent.be/ontologies/arseco/Face.owl#>.

                _:region owl:oneOf ?regionListOutput;
                         rdfs:subClassOf face:FaceRegion;
                         rdfs:subClassOf [owl:onProperty imreg:regionOf;
                                          owl:hasValue ?imageInput]."""."
```

A.5 Face recognition collaborator OWL-S description

```
@prefix : <http://multimedialab.elis.ugent.be/ontologies/
                                              arseco/FaceRecognitionService.owl#>.
@prefix dc: <http://purl.org/dc/elements/1.1/>.
@prefix Service: <http://www.daml.org/services/owl-s/1.1/Service.owl#>.
@prefix sd: <http://www.w3.org/2009/sparql/service-description#>.
@prefix Profile: <http://www.daml.org/services/owl-s/1.1/Profile.owl#>.
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#>.
@prefix owl2xml: <http://www.w3.org/2006/12/owl2-xml#>.
@prefix Process: <http://www.daml.org/services/owl-s/1.1/Process.owl#>.
@prefix xsd: <http://www.w3.org/2001/XMLSchema#>.
@prefix owl: <http://www.w3.org/2002/07/owl#>.
@prefix owl-s-sparql: <http://multimedialab.elis.ugent.be/ontologies/
                                                          arseco/owl-s-sparql.owl#>.
@prefix Expression: <http://www.daml.org/services/owl-s/1.1/generic/Expression.owl#>.

:FaceRecognitionService a Service:Service;
                   dc:description "Service that recognizes faces in a face region.";
                   Service:presents :FaceRecognitionProfile;
                   Service:describedBy :FaceRecognitionProcess;
                   Service:supports :FaceRecognitionServiceSparqlGrounding.

:FaceRecognitionProfile a Profile:Profile;
         Profile:serviceName "Face Recognition Collaborator";
         Profile:textDescription "Service that recognizes a face in a face region.";
         Profile:hasInput :Region;
         Process:hasOutput :Face;
         Profile:hasOutput :Person;
         Profile:has_process :FaceRecognitionProcess.

:FaceRecognitionProcess a Process:AtomicProcess;
                        Process:hasInput :Region;
                        Process:hasOutput :Face;
                        Process:hasOutput :Person;
                        Process:hasResult :FaceRecognitionResult.

:FaceRecognitionServiceSparqlGrounding a owl-s-sparql:SparqlServiceGrounding;
  owl-s-sparql:supportsQueryForm owl-s-sparql:SparqlQueryFormConstruct,
                                 owl-s-sparql:SparqlQueryFormSelect;
  owl-s-sparql:supportsSparqlVersion owl-s-sparql:SparqlVersionQuery1_0;
  sd:url <http://multimedialab.elis.ugent.be/ontologies/arseco/FaceRecognition/sparql>.

:Region a Process:Input;
  Process:parameterType "http://multimedialab.elis.ugent.be/ontologies/
                                              arseco/Face.owl#FaceRegion"^^xsd:anyURI.
```

```
:Face a Process:Output;
        Process:parameterType "http://multimedialab.elis.ugent.be/ontologies/
                                                    arseco/Face.owl#Face"^^xsd:anyURI.

:Person a Process:Output;
          Process:parameterType "http://xmlns.com/foaf/0.1/Person"^^xsd:anyURI.

:FaceRecognitionResult a Process:Result;
                       Process:hasEffect :FaceRecognitionEffect.

:FaceRecognitionEffect a owl-s-sparql:N3-Expression;
                       Expression:expressionBody
  """?regionInput <http://www.w3.org/2004/02/image-regions#regionDepicts> ?faceOutput.
     ?faceOutput <http://multimedialab.elis.ugent.be/ontologies/
                                       arseco/Face.owl#isFaceOf> ?personOutput.""".
```

Appendix B

Complexity Analysis of Adjacency Matrices

Section 4.4 introduced adjacency matrix composition, which uses binary matrices for efficient calculations. The time and memory complexity claims of Subsection 4.4.3 are verified in this appendix.

B.1 Structure

B.1.1 Bit vector

Before we consider binary matrices, we take a look at the one-dimensional case of binary arrays. Without special support, a traditional processor with word size w will store an array of n boolean values in a consecutive memory block of $n \cdot w$ bits. However, the theoretically required space is only n bits, since a boolean value can be represented by a single bit. Most processors work with a well-aligned memory structure, so we need to round up this number to the next multiple of w.

> **Definition B.1** A **bit vector** stores an array of n boolean values $[b_1, \dots, b_n]$ efficiently in $n' = w \cdot \lceil n/w \rceil$ bits. If we store the value of n separately, we can extend the size of the array from n to n', zeroing the values b_i where $n < i \leq n'$. Then, without loss of generality, we can interpret the array $[b_1, \dots, b_{n'}]$ as a sequence of w-digit binary representations of n'/w numbers $[(b_1 \dots b_w)_2, \dots, (b_{n'-w+1} \dots b_{n'})_2]$.

If n is predetermined, storing and retrieving a single bit takes $\Theta(1)$ time as its location is determined by a simple calculation. If n increases dynamically, we double the internal array size whenever n exceeds the number of bits that can currently be stored. This last operation takes $\Theta(n)$ time, but seldom needs to execute, which brings the total amortized time for this case also to $\Theta(1)$. Memory consumption is $\Theta(n)$ bits.

B.1.2 Bit matrix

Definition 4.5 describes an adjacency matrix as a square matrix whose elements are in $\mathbb{B}$. To provide fast access to entire rows *and* columns, we store the matrix twice: once as a set of bit vector rows and once as a set of bit vector columns.

Definition B.2 Bit matrix structure

A square bit matrix M of size n

$$M = \begin{array}{c} \\ \mathbf{r_1} \\ \vdots \\ \mathbf{r_i} \\ \vdots \\ \mathbf{r_n} \end{array} \begin{array}{c} \begin{array}{ccccc} \mathbf{c_1} & \cdots & \mathbf{c_j} & \cdots & \mathbf{c_n} \end{array} \\ \begin{pmatrix} m_{11} & \cdots & m_{1j} & \cdots & m_{1n} \\ \vdots & & \ddots & & \vdots \\ m_{i1} & \cdots & m_{ij} & \cdots & m_{in} \\ \vdots & & \ddots & & \vdots \\ m_{n1} & \cdots & m_{nj} & \cdots & m_{nn} \end{pmatrix} \end{array}$$

is represented by n row bit vectors

$$\mathbf{r_i} = \begin{pmatrix} m_{i1} & \cdots & m_{in} \end{pmatrix}$$

and n column bit vectors

$$\mathbf{c_j} = \begin{pmatrix} m_{1j} \\ \vdots \\ m_{nj} \end{pmatrix}.$$

The total amount of memory required is $\Theta(2 \cdot (n \cdot n)) = \Theta(n^2)$.

B.2 Operations

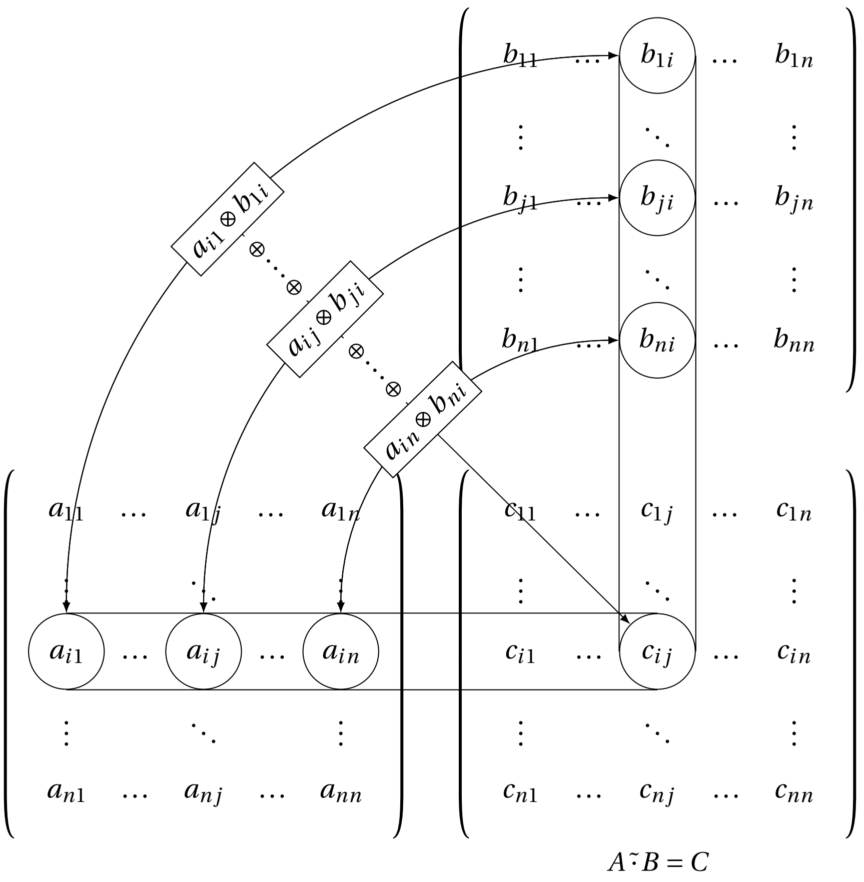
B.2.1 Row/column appending

Appending a new row – and thus, since the matrix is square, a new column – to a size n matrix requires appending a bit vector to the row and column sets, and appending an element to each of the n existing row and column bit vectors. This amounts to $\Theta(n)$ time.

B.2.2 Saturating multiplication

Regular matrix multiplication of two size n square matrices in $\mathbb{R}$ requires the calculation of n^2 elements, each of which is the sum of n products of elements. This results in a total time complexity of $\Theta(n^3)$.

The saturating multiplication of binary matrices $\tilde{\cdot}$ can be performed in a more efficient way. This calculation is equivalent to matrix multiplication in $\mathbb{R}$, replacing the product of elements by the binary *AND* operation $\otimes$ and their sum by the binary *OR* operation $\oplus$. This is represented schematically in Figure B.1.



b_{11} … b_{1i} … b_{1n}
b_{j1} … b_{ji} … b_{jn}
b_{n1} … b_{ni} … b_{nn}
$a_{i1} \oplus b_{1i}$
$\otimes \cdots \otimes$
$a_{ij} \oplus b_{ji}$
$\otimes \cdots \otimes$
$a_{in} \oplus b_{ni}$
a_{11} … a_{1j} … a_{1n}
a_{i1} … a_{ij} … a_{in}
a_{n1} … a_{nj} … a_{nn}
c_{11} … c_{1j} … c_{1n}
c_{i1} … c_{ij} … c_{in}
c_{n1} … c_{nj} … c_{nn}
$A \tilde{\cdot} B = C$

Figure B.1: Saturating multiplication of bit matrices

These computations can be sped up by taking advantage of the specific structure of bit vectors, partitioned in words with length w. Instead of *AND*-ing each element of the vector individually, we take advantage of the processor support for *AND*-ing them in groups of size w, which yields a new bit vector. The *OR* operation on the elements of this vector is performed by existential quantification: it is true if any of the n/w words with length w is non-zero. This brings the complexity down to $\Theta(n^3/w)$ with a low step cost, which is significant since the magnitudes of n and w will be similar in practice.

B.2.3 Saturating k^{th} power

The saturating k^{th} power of a bit matrix is calculated as the repeated saturated multiplication of the matrix with itself, applied k times. This subsequently requires $\Theta(k \cdot (n^3/w))$ time. Adjacency matrices use this operation with $k = n$ to calculate the extended connectivity matrix, which thus executes in $\Theta(n^4/w)$ time.

Appendix C

DVD

C.1 Contents

The enclosed DVD includes the following items:

- an **electronic version** of this document;
- the **source code** of the *Arseco* software and collaborators;
- a **video of the use case** of Chapter 7.

Bibliography

[1] Rakesh Agrawal, Tomasz Imieliński, and Arun Swami. “Mining association rules between sets of items in large databases”. In: *SIGMOD ’93: Proceedings of the 1993 ACM SIGMOD international conference on Management of data*. Washington, D.C., United States: ACM, 1993, pp. 207–216. ISBN: 0-89791-592-5. DOI: `http://doi.acm.org/10.1145/170035.170072`.

[2] Tim Berners-Lee. *Why the RDF model is different from the XML model*. Oct. 14, 1998. URL: `http://www.w3.org/DesignIssues/RDF-XML.html`.

[3] Tim Berners-Lee and Dan Connolly. *Notation3 (N3): A readable RDF syntax*. W3C Recommendation. Jan. 14, 2009. URL: `http://www.w3.org/TeamSubmission/n3/`.

[4] Tim Berners-Lee, Sandro Hawke, and Dan Connolly, eds. *Built-in properties in Cwm*. URL: `http://www.w3.org/2000/10/swap/doc/CwmBuiltins`.

[5] Tim Berners-Lee, James Hendler, and Ora Lassila. “The Semantic Web”. In: *Scientific American* 284.5 (2001), p. 34. ISSN: 00368733. URL: `http://search.ebscohost.com/login.aspx?direct=true&db=buh&AN=4328935&loginpage=Login.asp&site=ehost-live`.

[6] Tim Berners-lee et al. “N3Logic: A logical framework for the World Wide Web”. In: *Theory and Practice of Logic Programming* 8.3 (2008), pp. 249–269. ISSN: 1471-0684. DOI: `http://dx.doi.org/10.1017/S1471068407003213`.

[7] Christian Bizer, Tom Heath, and Tim Berners-Lee. “Linked Data – The Story So Far”. In: *International Journal on Semantic Web and Information Systems (IJSWIS)* 5.3 (2009). URL: `http://tomheath.com/papers/bizer-heath-berners-lee-ijswis-linked-data.pdf`.

[8] Christian Bizer et al. “DBpedia - A crystallization point for the Web of Data”. In: *Web Semant.* 7.3 (2009), pp. 154–165. ISSN: 1570-8268. DOI: `http://dx.doi.org/10.1016/j.websem.2009.07.002`.

[9] Gary Bradski and Adrian Kaehler. *Learning OpenCV: Computer Vision with the OpenCV Library*. Cambridge, MA: O’Reilly, 2008.

[10] Dan Brickley and Ramanathan V. Guha, eds. *RDF Vocabulary Description Language 1.0: RDF Schema*. W3C Recommendation. Feb. 10, 2004. URL: `http://www.w3.org/TR/2004/REC-rdf-schema-20040210/`.

[11] Kendall Grant Clark, Lee Feigenbaum, and Elias Torres, eds. *SPARQL Protocol for RDF*. W3C Recommendation. Jan. 15, 2008. URL: `http://www.w3.org/TR/rdf-sparql-protocol/`.

[12] Daniel D. Corkill. "Blackboard Systems". In: *AI Expert* 6.9 (Sept. 1991), pp. 40–47. URL: `http://bbtech.com/papers/ai-expert.pdf`.

[13] *DBpedia*. URL: `http://dbpedia.org/`.

[14] Jos De Roo. *Euler Proof Mechanism*. URL: `http://eulersharp.sourceforge.net/`.

[15] Kutluhan Erol, James Hendler, and Dana S. Nau. "HTN planning: Complexity and expressivity". In: *Proceedings of the Twelfth National Conference on Artificial Intelligence (AAAI-94)*. Vol. 2. 1994, pp. 1123–1128.

[16] Richard E. Fikes and Nils J. Nilsson. "STRIPS: A new approach to the application of theorem proving to problem solving". In: *Artificial Intelligence* 2.3-4 (1971), pp. 189–208. DOI: `10.1016/0004-3702(71)90010-5`.

[17] Ian Forrester. *BBC Backstage SPARQL Endpoint*. June 10, 2009. URL: `http://www.bbc.co.uk/blogs/bbcbackstage/2009/06/bbc-backstage-sparql-endpoint.shtml`.

[18] Martin Fowler. *Inversion of Control Containers and the Dependency Injection pattern*. Jan. 4, 2004. URL: `http://martinfowler.com/articles/injection.html`.

[19] Roberto García and Òscar Celma. "Semantic Integration and Retrieval of Multimedia Metadata". In: *Proceedings of the 5th International Workshop on Knowledge Markup and Semantic Annotation (SemAnnot 2005)*. 2005. URL: `http://rhizomik.net/content/roberto/papers/rgocsemannot2005.pdf`.

[20] Michael R. Genereth, ed. *Knowledge Interchange Format*. Draft Proposed American National Standard. URL: `http://logic.stanford.edu/kif/dpans.html`.

[21] Matthias De Geyter and Peter Soetens. "A Planning Approach to Media Adaptation within the Semantic Web." In: *DMS*. 2005, pp. 129–134.

[22] Martin Gudgin et al., eds. *SOAP Version 1.2 Part 1: Messaging Framework (Second Edition)*. W3C Recommendation. Apr. 27, 2007. URL: `http://www.w3.org/TR/2007/REC-soap12-part1-20070427/`.

[23] Oktie Hassanzadeh and Mariano Consens. "Linked Movie Data Base". In: *Proceedings of the WWW2009 workshop on Linked Data on the Web (LDOW2009)*. Madrid, Spain 2009.

[24] Ian Horrocks et al. *SWRL: A Semantic Web Rule Language Combining OWL and RuleML.* W3C Member Submission. May 21, 2004. URL: http://www.w3.org/Submission/SWRL/.

[25] Anna Hristoskova and Dieter Moeyersoon. "Dynamische compositie van medische web services in OWL-S". MA thesis. Belgium: Ghent University, 2008. URL: http://lib.ugent.be/fulltxt/thesis/2012_Hristoskova.pdf.

[26] Anna Hristoskova, Bruno Volckaert, and Filip De Turck. "Dynamic Composition of Semantically Annotated Web Services through QoS-Aware HTN Planning Algorithms". In: *ICIW '09: Proceedings of the 2009 Fourth International Conference on Internet and Web Applications and Services.* Washington, DC, USA: IEEE Computer Society, 2009, pp. 377–382. ISBN: 978-0-7695-3613-2.

[27] Jane Hunter. "Adding Multimedia to the Semantic Web – Building an MPEG-7 Ontology". In: *International Semantic Web Working Symposium (SWWS).* 2001, pp. 261–281.

[28] Graham Klyne and Jeremy J. Carrol, eds. *Resource Description Framework (RDF): Concepts and Abstract Syntax.* W3C Recommendation. Feb. 10, 2004. URL: http://www.w3.org/TR/2004/REC-rdf-concepts-20040210/.

[29] Kenneth J. Laskey et al., eds. *Uncertainty Reasoning for the World Wide Web.* W3C Incubator Group Report. Mar. 31, 2008. URL: http://www.w3.org/2005/Incubator/urw3/XGR-urw3-20080331/.

[30] David Martin et al. *OWL-S: Semantic Markup for Web Services.* W3C Member Submission. Nov. 22, 2004. URL: http://www.w3.org/Submission/OWL-S/.

[31] J. M. Martinez, ed. *MPEG-7 Overview.* Oct. 2004. URL: http://mpeg.chiariglione.org/standards/mpeg-7/mpeg-7.htm.

[32] Deborah L. McGuinness and Frank van Harmelen, eds. *OWL Web Ontology Language Overview.* W3C Recommendation. Feb. 10, 2004. URL: http://www.w3.org/TR/2004/REC-owl-features-20040210/.

[33] *Media Fragments Working Group.* URL: http://www.w3.org/2008/WebVideo/Fragments/.

[34] Simon Parsons. "Current Approaches to Handling Imperfect Information in Data and Knowledge Bases". In: *IEEE Trans. on Knowl. and Data Eng.* 8.3 (1996), pp. 353–372. ISSN: 1041-4347. DOI: http://dx.doi.org/10.1109/69.506705.

[35] Randall Perrey and Mark Lycett. "Service-Oriented Architecture". In: *IEEE/IPSJ International Symposium on Applications and the Internet Workshops* (2003), p. 116. DOI: http://doi.ieeecomputersociety.org/10.1109/SAINTW.2003.1210138.

[36] Eric Prud'hommeaux and Andy Seaborne, eds. *SPARQL Query Language for RDF.* W3C Recommendation. Jan. 15, 2008. URL: http://www.w3.org/TR/rdf-sparql-query/.

[37] Domenico Redavid, Luigi Iannone, and Terry Payne. "OWL-S Atomic services composition with SWRL rules". In: *Proceedings of the 4th Italian Semantic Web Workshop.* 2007. URL: http://eprints.ecs.soton.ac.uk/15658/.

[38] Peter Van Roy and Seif Haridi. *Concepts, Techniques, and Models of Computer Programming.* Cambridge, MA, USA: MIT Press, 2004. ISBN: 0-262-22069-5.

[39] John R. Smith and Peter Schirling. "Metadata Standards Roundup". In: *IEEE MultiMedia* 13 (2006), pp. 84–88. ISSN: 1070-986X. DOI: http://doi.ieeecomputersociety.org/10.1109/MMUL.2006.34.

[40] Peter Soetens and Matthias De Geyter. "Applying Domain Knowledge to Multi-Step Media Adaptation Based on Semantic Web Services". In: *Workshop on Image Analysis for Multimedia Interactive Services (WIAMIS 2005).* Montreux, Switzerland 2005.

[41] Peter Soetens and Matthias De Geyter. "Multi-step media adaptation: implementation of a knowledge-based engine". In: *WWW '05: Special interest tracks and posters of the 14th international conference on World Wide Web.* Chiba, Japan: ACM, 2005, pp. 986–987. ISBN: 1-59593-051-5.

[42] Joshua Tauberer. *SemWeb RDF Library for C#/.Net.* URL: http://razor.occams.info/code/semweb/.

[43] Jelena Tešić. "Metadata Practices for Consumer Photos". In: *IEEE MultiMedia* 12.3 (2005), pp. 86–92. ISSN: 1070-986X. DOI: http://dx.doi.org/10.1109/MMUL.2005.50.

[44] Chrisa Tsinaraki, Polydoros Panagiotis, and Christodoulakis Stavros. "Interoperability support for Ontology-based Video Retrieval Applications". In: *Proceedings of 3rd International Conference on Image and Video Retrieval (CIVR 2004).* 2004, pp. 582–591.

[45] Steven Verstockt et al. "Actor recognition for interactive querying and automatic annotation in digital video". In: *IASTED International conference on Internet and Multimedia Systems and Applications, 13th, Proceedings.* Honolulu, HI, USA: ACTA Press, 2009, pp. 149–155.

[46] Paul Viola and Michael J. Jones. "Robust Real-Time Face Detection". In: *International Journal of Computer Vision* 57.2 (May 1, 2004), pp. 137–154. ISSN: 0920-5691. DOI: 10.1023/B:VISI.0000013087.49260.fb.

[47] Todd Gregory Williams, ed. *SPARQL 1.1 Service Description.* SPARQL Working Draft. Oct. 22, 2009. URL: http://www.w3.org/TR/2009/WD-sparql11-service-description-20091022/.

[48] Todd Gregory Williams and Ivan Mikhailov, eds. *SPARQL Service Description.* SPARQL Working Group. 2009. URL: http://www.w3.org/2009/sparql/wiki/Feature:ServiceDescriptions.

Gebruik van Semantisch-webtechnologieën voor (Semi-)Automatische Metadatageneratie bij Multimediale Data

Ruben Verborgh
Promotor: prof. dr. ir. Rik Van de Walle
Begeleiders: dr. Davy Van Deursen en Erik Mannens

Academiejaar 2009–2010

ABSTRACT — *Deze masterproef onderzoekt de toepassing van semantisch-webtechnologieën voor automatische metadatageneratie en -annotatie bij multimediale data. We beschrijven de architectuur van een algemene semantische probleemoplosser die onafhankelijke algoritmes gebruikt om deeltaken uit te voeren. Een holistische visie op metadataverzoeken maakt het aanpakken van complexere problemen mogelijk en laat de algoritmes interageren met de oplossing in wording.*

1 Inleiding

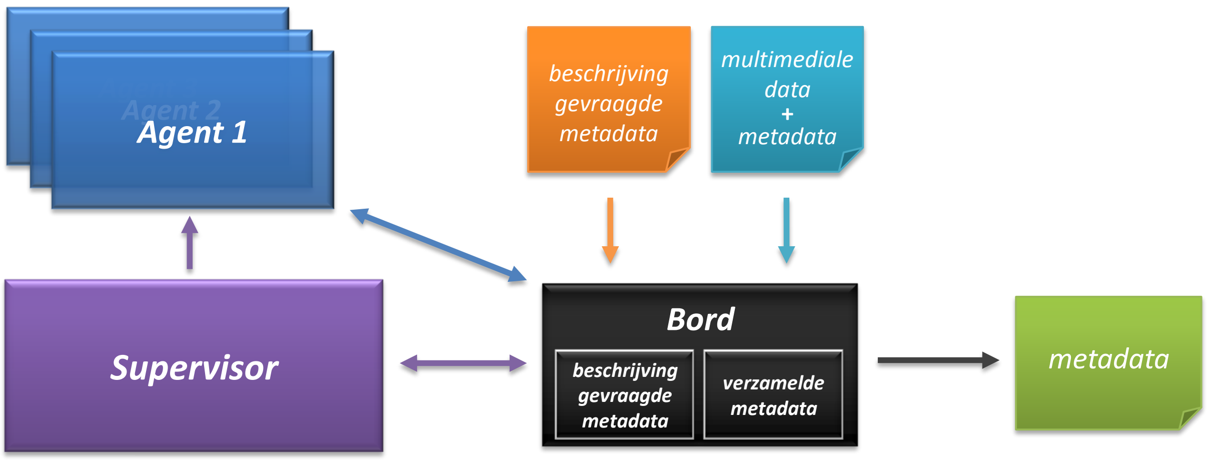
Gedurende de voorbije jaren kenden de productie en consumptie van multimediale data een ongeziene groei dankzij het internet. Doorzoeken van deze data blijkt echter niet zo eenvoudig als het doorzoeken van tekst. Metadata [1] biedt hiervoor een oplossing, maar helaas verhoogt de aanmaak hiervan de totale productiekost aanzienlijk. Automatisering van dit proces kan deze drempel verlagen. Momenteel bestaan er algoritmes voor specifieke verwerkingstaken, maar een intelligente totaalvisie op het te annoteren voorwerp ontbreekt.

In deze masterproef onderzoek ik hoe we semantisch-webtechnologieën kunnen inschakelen voor de aanmaak van metadata. Hiervoor ontwerpen en implementeren we een softwareplatform dat deze techniek demonstreert op afbeeldingen, waarbij we ervoor zorgen dat het concept uitbreidbaar is naar andere domeinen.

De volgende sectie belicht de architectuur van een platform voor metadatageneratie. De secties 3 tot en met 5 stellen een algemene semantische probleemoplosser voor, die we implementeren in sectie 6. De toepassing op multimediale data komt aan bod in het praktijkvoorbeeld van sectie 7. Tot slot bekijken we mogelijkheden voor toekomstig onderzoek in sectie 8 en sluiten af met een algehele conclusie in sectie 9.

2 Architectuur

Metadatageneratie is een proces met als invoer een *metadataverzoek* voor een specifiek *voorwerp* en als uitvoer de gevonden *metadata*. Wegens de vereiste veelzijdige inzetbaarheid, kan de processor die dit realiseert duidelijk geen monolithische structuur hebben. We splitsen de coördinerende en uitvoerende taken, en onderscheiden een ***supervisor*** die verschillende ***zelfstandige agenten*** aanstuurt. Deze architectuur, weergegeven in Figuur 1, steunt op het *blackboard*-patroon [2], waarbij de supervisor en de agenten via een ***bord*** communiceren in een gemeenschappelijk uitwisselingsformaat. Hierbij opteren we voor het door machines interpreteerbare *RDF*-formaat [3] omwille van de rigoureuze semantische structuur, het bestaan van redeneermechanismen en de reeds beschikbare kennis.


Agent 2
Agent 1
beschrijving gevraagde metadata
multimediale data + metadata
Supervisor
Bord
beschrijving gevraagde metadata
verzamelde metadata
metadata

Figuur 1: Architectuur van de processor, gebaseerd op het *blackboard*-patroon

Volgende componenten maken deel uit van het systeem:

- **Agenten** vervullen specifieke taken, gerelateerd aan het verzoek of deelverzoeken hiervan. We kunnen hen opdelen in verschillende klassen zoals kenmerkextractie- en redeneeralgoritmes. Ze gaan vergezeld van een formele beschrijving van capaciteiten en vereisten. Hun brede toepassingsgebied maakt een verregaande interoperabiliteit noodzakelijk.
- De **supervisor** coördineert het volledige proces. Dit begint met het opstellen van een plan, gevolgd door de uitvoering hiervan. Ten slotte formuleert de supervisor een antwoord op het oorspronkelijke verzoek.
- Het **bord** is een eenvoudige RDF-opslagplaats waarmee de verschillende entiteiten informatie uitwisselen. De supervisor kan de aanwezige informatie verrijken door semantische redeneerprocessen. Wanneer agenten hun eigen werking verbeteren aan de hand van uitvoer van vorige agenten, ontstaat ***semantische terugkoppeling***.

De voorgestelde architectuur biedt een solide en flexibele basis voor een algemeen semantisch probleemoplossend platform.

3 Agenten

3.1 Interactie

Als input- en outputformaat voor agenten, gebruiken we eveneens RDF wegens de flexibiliteit van deze formele datarepresentatie en de brede interoperabiliteit. Daarnaast gebruiken we SPARQL als bevragingstaal [4] en protocol [5], omdat dit ons toelaat agenten transparant en flexibel aan te roepen. Onderstaand voorbeeld toont een interactie.

Voorbeeld 1 Interactie met een agent die het quotiënt van gehele getallen berekent

Aanvraag

```
PREFIX : <http://example.org/SparqlRequest.owl#>
PREFIX math: <http://example.org/Math.owl#>
CONSTRUCT { math:solution math:hasValue ?quotient. }
WHERE {
  [a :Request;
   :method "Divide";
   :input  [a :ParameterValue; :bindsParameter "dividend"; :boundTo 15],
           [a :ParameterValue; :bindsParameter "divisor";  :boundTo 5];
   :output [a :ParameterValue; :bindsParameter "quotient"; :boundTo ?quotient]]
}
```

Uitvoer

```
math:solution math:hasValue 3.
```

Een algoritme met RDF-invoer en -uitvoer kunnen we eenvoudig omzetten in een agent. Eerst extraheren we de invoerwaarden uit de aanvraag, waaruit de agent vervolgens uitvoer genereert. Vervolgens plaatsen we invoer en uitvoer samen in een virtuele RDF-entiteit van type `Request`, waarop een SPARQL-eenheid de oorspronkelijke aanvraag uitvoert.

3.2 Beschrijving

Vermits agenten gewone SPARQL-eindpunten zijn en bijgevolg dus webservices, kunnen we ze beschrijven met *OWL-S* [6], een vaak gebruikte specificatie hiervoor. Beschrijvingen bestaan uit volgende elementen:

- een **profiel**, een beknopte omschrijving van capaciteiten en vereisten;
- een **model** met verdere details over de capaciteiten en vereisten, voldoende uitgebreid om composities op te stellen;
- een **verankering** die het protocol beschrijft, in dit geval de SPARQL-parameters.

Deze beschrijving drukt de werking van een agent zeer formeel uit, zodat het compositieproces relevante agenten kan inschakelen.

4 Compositie

De mogelijkheden van individuele agenten volstaan niet voor complexere problemen. Compositiealgoritmen combineren agenten tot uitvoeringsplannen om aan geavanceerdere verzoeken te voldoen.

Een compositie bestaat uit aanroepen van agenten met bepaalde parameters. Deze parameters zijn ofwel constante waarden, ofwel het resultaat van vorige aanroepen; in dit laatste geval spreken we van een *afhankelijkheid* tussen beide.

4.1 Componeren met verbindingsmatrices

Een eerste compositiealgoritme stelt een *verbindingsmatrix* op van alle aanwezig agenten, die weergeeft of de aanroep van de ene agent de aanroep van een andere mogelijk maakt. Hiervoor beschikken we over een *OWL-S-koppelaar* die de compatibiliteit van twee OWL-S-beschrijvingen kan nagaan.

Door deze matrix herhaaldelijk met zichzelf te vermenigvuldigen, krijgen we een beeld van de mogelijke aanroepketens van agenten. Om een compositie op te stellen, zetten we eerst de invoer en de aanvraag om in respectievelijk een begin- en eindservice. Vervolgens gaan we na aan welke agenten we deze services kunnen koppelen. Ten slotte lezen we in de verbindingsmatrix hoe deze agenten met elkaar verbonden zijn.

Dit algoritme vindt niet steeds een oplossing, zelfs als deze toch bestaat. Verbindingsmatrices creëren immers enkel composities die uit een aaneengesloten keten bestaan.

4.2 Componeren met semantische redenering

Als we de OWL-S-beschrijvingen echter omzetten in N3-regels [7], kan een semantisch redeneerprogramma [8] de compositie opstellen. Het voordeel ligt in het feit dat dit redeneerprogramma het compositieprobleem in zijn geheel beschouwt en hierop semantische kennis kan toepassen. Deze holistische benadering maakt complexere composities mogelijk. Het algoritme functioneert als volgt:

1. We **vertalen** de OWL-S-beschrijvingen naar N3 regels, waarbij het antecedent bestaat uit invoer en precondities, en het consequent uit uitvoer en postcondities;
2. Het redeneerprogramma voert **deducties** uit, waarbij we de beginservice als gegeven veronderstellen en de eindservice als doelstelling nemen;
3. We **reconstrueren** de compositie door te achterhalen welke regels het redeneerprogramma gebruikte en deze om te zetten in aanroepen van agenten.

Dit algoritme is in staat om zo goed als alle composities te realiseren, mits voldoende relevante semantische kennis en agenten beschikbaar zijn.

5 Supervisie

Een supervisor is verantwoordelijk voor de selectie van het gepaste plan en de uitvoering hiervan. Hij geeft de vooruitgang weer, biedt mogelijkheden om fouten te herstellen en eindigt met het formuleren van een antwoord op de aanvraag.

Selectie In het selectieproces kunnen verschillende parameters een rol spelen, zoals *kost, nauwkeurigheid, prestatie, beschikbaarheid* en *volledigheid.* In de praktijk moeten we steeds een afweging maken tussen verschillende factoren, waarbij de parameterwaarden mogelijk niet of slechts gedeeltelijk bepaald zijn.

Uitvoering De supervisor interpreteert het uitvoeringsplan als een declaratief programma [9] en zet dit om in een feitelijke uitvoering. De onderlinge afhankelijkheden bepalen de volgorde van de aanroepen. Alle resultaatwaarden die de agenten genereren, vormen koppels met de veranderlijke waaraan ze zijn toegewezen. We plaatsen hen in de veranderlijkenopslag, die blijft bestaan tijdens de volledige duur van het proces.

Weergave Als we begrijpen waarom tijdens het proces bepaalde keuzes gemaakt werden, kunnen we de gebruikte kennis of agenten aanpassen. De uitvoering van eenvoudige composities kunnen we weergeven als een chronologisch overzicht van aanroepen. Complexere composities komen beter tot hun recht in een hiërarchische weergave, waarbij we duidelijker het verband zien tussen de aanroepen en de te bereiken doelen.

Herstel Het foutherstelproces bestaat uit drie delen:

1. De supervisor **detecteert** de fout: ofwel trad er een storing op in de agent, ofwel gaf de agent een antwoord dat niet aan de verwachtingen voldoet;
2. We bepalen de **impact**, identificeren het breekpunt en kijken welke delen van de compositie en het uiteindelijke antwoord getroffen zijn;
3. We stellen een **herstelplan** op dat begint vanaf de huidige bordinhoud en eindigt in het herstelpunt. Als de impact een groot deel van de compositie bestrijkt, kunnen we ook een volledig nieuwe compositie kiezen.

Bij het opstellen kunnen we gebruik maken van reeds verzamelde informatie en aanvullende semantische kennis, zodat het herstelplan meer gericht is op de concrete situatie dan het originele plan.

Antwoord Aan het einde van het proces formuleert de supervisor een antwoord. Dit kan bestaan uit enkel de gevraagde uitvoer, de volledige inhoud van het bord of een combinatie van beide. Voor metadatageneratie kan het interessant zijn om de inhoud van het bord te gebruiken omdat tussenresultaten – hoewel niet expliciet gevraagd – ook waardevolle annotaties kunnen bevatten.

6 Implementatie

We implementeerden de concepten uit de vorige secties in een softwareplatform genaamd *Arseco*, dat de theoretische basis nauwlettend volgt. Het platform bestaat uit een bibliotheek voor agenten en een semantische probleemoplosser, met als architecturale kenmerken:

- een **hoge aanpasbaarheid** van de componenten om onderzoek naar de impact van wijzigingen en de inpasbaarheid in andere situaties te vergemakkelijken;
- **brede interoperabiliteit** van agenten, zodat deze transparant kunnen samenwerken onafhankelijk van hun concrete implementatie;
- **herbruikbaarheid**: we dienen componenten zo onafhankelijk te maken van concrete probleemsituaties om hergebruik te maximaliseren.

6.1 Agenten

We ontwikkelden een raamwerk voor de functionaliteit van agenten, dat inzetbaar is op twee manieren. Enerzijds kan het dienst doen als een transformator voor bestaande algoritmes, door deze via configuratie te vertalen naar een SPARQL-eindpunt. Anderzijds biedt het een bibliotheek aan met vaak voorkomende taken zoals inlezen van RDF en verwerken van SPARQL. We voorzien implementaties in C++ en C#, en een opdrachtpromptversie om een breed scala aan algoritmes te bereiken.

Verder implementeerden we ook een HTTP-webserver die verschillende agenten kan herbergen en het SPARQL-protocol ondersteunt. Daarnaast bevat deze ook functies om agenten via een webbrowser te bevragen.

6.2 Supervisor

We hebben de supervisorimplementatie zeer algemeen gehouden en afgeschermd van specifieke compositie- en kenniscomponenten. Dit geeft ons de mogelijkheid om experimenten met verschillende concrete versies uit te voeren. We implementeerden de ondersteuning voor OWL-S-beschrijvingen, en compositiealgoritmes met verbindingsmatrices en semantische redenering, beiden compatibel met verschillende redeneerprogramma's.

Dankzij een flexibele configuratiestrategie, die gebruikt maakt van componentinjectie, kunnen we het platform dynamisch wijzigen en agenten of kennisbronnen eenvoudig aan- en uitschakelen door middel van een configuratiebestand.

Het is belangrijk op te merken dat het geïmplementeerde platform een algemene semantische probleemoplosser vormt. Gebruik in concrete domeinen vereist de toevoeging van relevante agenten en semantische kennis. In de volgende sectie bekijken we een concreet voorbeeld voor metadatageneratie.

7 Praktijkvoorbeeld

7.1 Probleemstelling

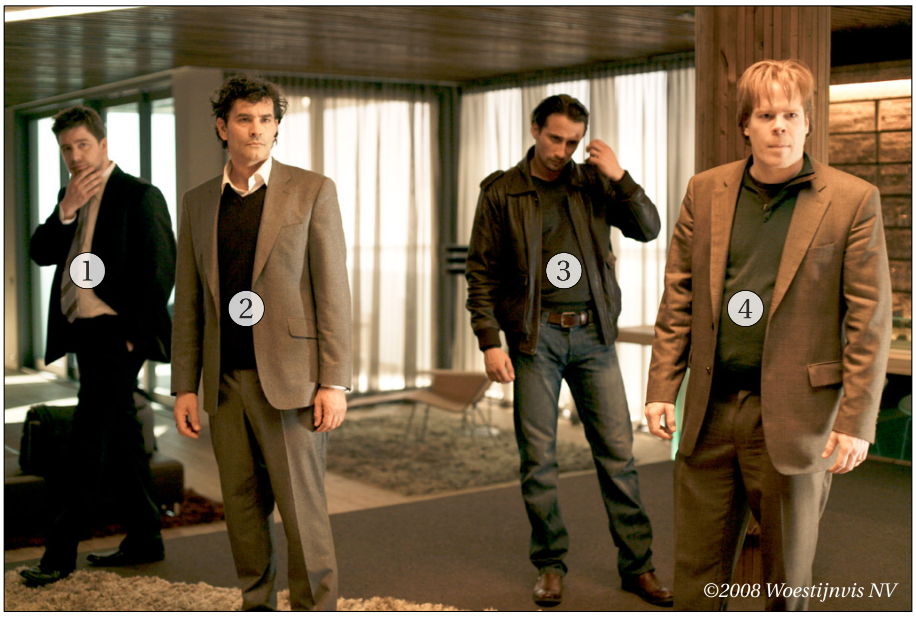
Veronderstel een digitaal fotoarchief met enkel afbeeldingsdata, waarin we beroemde personen willen herkennen. We beschikken over de volgende agenten:

- een implementatie van het **gezichtsdetectiealgoritme** van Viola-Jones [10], dat gebieden met een gezicht opspoort in een afbeelding;
- een implementatie van het **gezichtsherkenningsalgoritme** van Verstockt et al. [11], dat een gezicht in een gebied herkent aan de hand van een voorbeeldreeks.

Daarnaast hebben we toegang tot volgende semantische kennis:

- ontologieën en regels voor afbeelding, gebied en gezicht;
- kennis uit het semantisch web, in het bijzonder over beroemdheden.

In dit praktijkvoorbeeld spitsen we onze aandacht op de afbeelding `Loft.jpg` met vier acteurs, weergegeven in Figuur 2. Automatische persoonherkenning wordt gehinderd door lensonscherpte (acteur 1), de overlap met de rechterhand (acteur 1) en schaduw (acteur 3). We bekijken hoe de semantische probleemoplosser deze afbeelding verwerkt en hoe hij de voorgenoemde problemen probeert te overwinnen.



Figuur 2: Filmfoto met vier personen

7.2 Compositie

De supervisor zet de invoer (*de afbeelding*) en het verzoek (*alle personen herkennen*) om in respectievelijk een begin- en eindservice. Met semantische redeneersoftware vinden we een compositie die afbeelding, gebied en persoon met elkaar in verband brengt: een aanroep van de gezichtsdetectieagent gevolgd door een aanroep van de gezichtsherkenner.

7.3 Uitvoering

Gezichtsdetectie De supervisor roept de gezichtsdetectieagent aan met de parameter `Loft.jpg`, waarna de agent de vier gebieden correct identificeert en antwoordt. Deze gebieden komen terecht in de veranderlijkenopslag.

Gezichtsherkenning Vervolgens is het de beurt aan de gezichtsherkenningsagent, aangeroepen door de supervisor voor elk van de vier gebieden. Voor acteurs 2 (*Koen De Bouw*) en 4 (*Bruno Vanden Broucke*) vinden we het correcte resultaat. De herkenning van acteurs 1 en 3 faalt echter, waardoor de veranderlijkenopslag een onvolledig resultaat bevat. De supervisor detecteert deze fout.

Foutherstel Eerst probeert de supervisor het bord semantisch te verrijken. Hiervoor raadpleegt het DBpedia met de vraag tot meer informatie over de aanwezige entiteiten. Vermits de gevonden personen acteurs zijn, geeft DBpedia informatie over de films waarin zij meespeelden en wie verder nog meewerkte aan deze films. Vervolgens kan de supervisor afleiden dat, gezien het feit dat zij samenwerkten, collega's van de gevonden acteurs statistisch een hogere kans vertonen om samen op de foto te staan. Hierop vult hij de compositie aan met een aanroep van de gezichtsherkenningsagent, waarbij we als extra parameter meegeven welke personen we kunnen verwachten.

Gezichtsherkenning (bis) De gezichtsherkenningsagent gebruikt deze parameter om de kansen van bepaalde gezichten hoger in te schatten. Acteur 3 (*Matthias Schoenaerts*) herkent hij nu correct. Voor acteur 1 vindt het algoritme twee mogelijkheden, met ongeveer gelijke waarschijnlijkheid, en geeft beide terug: *Bruno Vanden Broucke* en *Koen De Graeve*.

Semantische deductie In principe zou nu opnieuw foutdetectie optreden. De supervisor bevat echter voldoende semantische kennis om te besluiten dat de foto onmogelijk tweemaal dezelfde persoon kan bevatten. Acteur 4 werd reeds herkend als *Bruno Vanden Broucke*, bijgevolg moet acteur 1 dus *Koen De Graeve* zijn.

Antwoord De supervisor formuleert een antwoord op basis van de inhoud van het bord. Afhankelijk van wat gevraagd is, geven we ofwel enkel het resultaat terug (de vier namen van deze acteurs), ofwel de volledige inhoud met verdere informatie over de acteurs. De verdere doeleinden van deze gegevens bepalen de keuze.

In dit praktijkvoorbeeld komt de meerwaarde van semantische kennis duidelijk naar voren. De relatie tussen de personen op de foto zorgde voor een volledige annotatie, wat individuele algoritmen niet kunnen bereiken omdat zij enkel op de hoogte zijn van hun eigen taak. De semantische terugkoppeling liet toe om de werking aan te passen aan de context.

8 Onderzoekspotentieel

Deze masterproef vormt slechts een inleiding tot de mogelijkheden van semantische probleemoplossing. Onderwerpen voor verder onderzoek omvatten volgende punten.

Kwaliteitsmaten Vooraleer we het platform inzetten in praktische situaties, moeten we de eigenschappen kunnen vergelijken met andere systemen. In het bijzonder willen we de meerwaarde van de semantische technologieën kwantificeren. Hiervoor hebben we kwaliteitsmaten nodig zoals *correctheid*, *volledigheid* en *prestatie*.

Imperfecte informatie Tot nu veronderstelden we dat de verkregen informatie een volledig waarheidsgehalte had. De resultaten van agenten zijn echter vaak onderhevig aan imperfecties, die we kunnen onderverdelen in orthogonale categorieën [12]:

- **onnauwkeurigheid:** de informatie is minder nauwkeurig dan verwacht;
- **vaagheid:** de informatie is inherent onscherp omschreven;
- **onzekerheid:** de waarheidswaarde is minder dan 100%;
- **onvolledigheid:** gedeelten van de informatie ontbreken;
- **inconsistentie:** de informatie spreekt zichzelf tegen.

Hoewel we dit tot op zekere hoogte kunnen simuleren met klassieke structuren, vragen sommige problemen om een meer rigoureuze omgang met imperfecties.

Kennismodellering Zoals reeds aangehaald, blijkt de aanwezigheid van relevante semantische kennis vaak een struikelblok. Verder hebben we ook het probleem dat sommige kennis zich moeilijk in ontologieën laat gieten, waardoor regels nodig zijn. Zij structureren kennis echter in een te specifieke vorm, wat hergebruik hindert. Eventueel kunnen we op zoek gaan naar automatische inductie van regels uit voorgaande probleemoplossingen.

Andere toepassingsdomeinen De uitbreiding naar andere toepassingsdomeinen stelt ons voor specifieke uitdagingen. In het geval van video bijvoorbeeld, zal de hoeveelheid aanroepen enkele grootteordes hoger liggen. Zeker voor toepassingen in ware tijd wordt het dan moeilijk om een overzicht te houden op het proces. Voor voldoende prestaties dienen we de supervisor uit te rusten met methodes voor parallelle of gegroepeerde uitvoering.

Gebruikersinteractie Tot slot kunnen we onderzoek richten op het inschakelen van mensen in het probleemoplossend proces. Dit kan bijvoorbeeld ter evaluatie van oplossingen of als assistentie bij taken waarin mensen beter presteren dan courante algoritmes. Er kan een gebruikersprofiel ontstaan, waaruit het platform leert om zijn werking te verbeteren en zich te richten op specifieke gebruikersnoden.

9 Conclusie

In deze masterproef onderzochten we hoe we semantische technologieën kunnen gebruiken voor metadatageneratie bij multimediale data. Hiervoor creëerden we een algemene semantische probleemoplosser met een *blackboard*-architectuur. Verschillende zelfstandige agenten verrichten een specifieke taak en gedragen zich naar de buitenwereld toe als een SPARQL-eindpunt, beschreven in OWL-S. Compositiealgoritmen schakelen deze agenten aaneen tot een plan dat een complex verzoek oplost. Een supervisor staat in voor de uitvoering van dit plan en de foutafhandeling ervan.

We bekeken een software-implementatie van dit platform, waarop een praktijkvoorbeeld met fotoannotatie werd toegepast. Daarnaast onderzochten we mogelijke paden voor toekomstig onderzoek in het gebied van semantische probleemoplossing.

Bibliografie

[1] J. R. Smith and P. Schirling, "Metadata standards roundup," *IEEE MultiMedia*, vol. 13, pp. 84–88, 2006.

[2] D. D. Corkill, "Blackboard systems," *AI Expert*, vol. 6, no. 9, pp. 40–47, Sep. 1991. [Online]. Available: http://bbtech.com/papers/ai-expert.pdf

[3] G. Klyne and J. J. Carrol. (2004, Feb.) Resource description framework (RDF): Concepts and abstract syntax. W3C Recommendation. [Online]. Available: http://www.w3.org/TR/2004/REC-rdf-concepts-20040210/

[4] E. Prud'hommeaux and A. Seaborne. (2008, Jan.) SPARQL query language for RDF. W3C Recommendation. [Online]. Available: http://www.w3.org/TR/rdf-sparql-query/

[5] K. G. Clark, L. Feigenbaum, and E. Torres. (2008, Jan.) SPARQL protocol for RDF. W3C Recommendation. [Online]. Available: http://www.w3.org/TR/rdf-sparql-protocol/

[6] D. Martin *et al.* (2004, Nov.) OWL-S: Semantic markup for web services. W3C Member Submission. [Online]. Available: http://www.w3.org/Submission/OWL-S/

[7] T. Berners-lee *et al.*, "N3Logic: A logical framework for the World Wide Web," *Theory and Practice of Logic Programming*, vol. 8, no. 3, pp. 249–269, 2008.

[8] J. De Roo. Euler proof mechanism. [Online]. Available: http://eulersharp.sourceforge.net/

[9] P. V. Roy and S. Haridi, *Concepts, Techniques, and Models of Computer Programming.* Cambridge, MA, USA: MIT Press, 2004.

[10] P. Viola and M. J. Jones, "Robust real-time face detection," *International Journal of Computer Vision*, vol. 57, no. 2, pp. 137–154, May 2004.

[11] S. Verstockt *et al.*, "Actor recognition for interactive querying and automatic annotation in digital video," in *IASTED International conference on Internet and Multimedia Systems and Applications, 13th, Proceedings.* Honolulu, HI, USA: ACTA Press, 2009, pp. 149–155.

[12] S. Parsons, "Current approaches to handling imperfect information in data and knowledge bases," *IEEE Trans. on Knowl. and Data Eng.*, vol. 8, no. 3, pp. 353–372, 1996.

Google
the droids we're looking for
Google Search
I'm Feeling
Advertising Programs - Business Solutions - Abo
© 2009 - Privac
oogle - Mozilla F

Foto: © Stéfan Le Dû

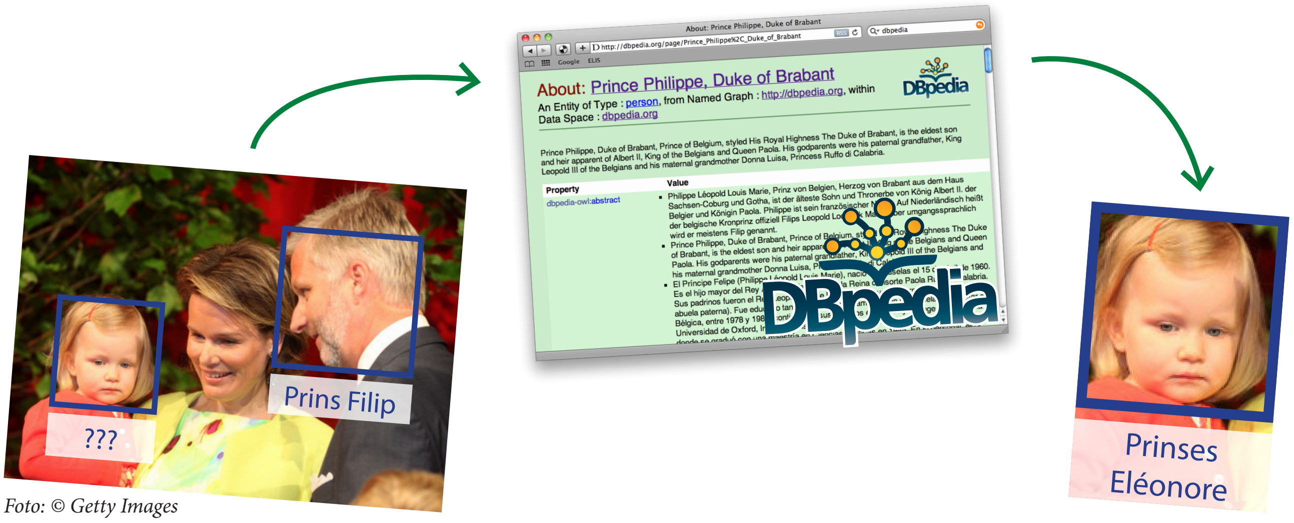
Intelligente internet helpt je slimmer zoeken

Zoek je de laatste nieuwtjes over Johnny Depp of Keira Knightley? Google is je vriend: in een fractie van een seconde schotelt de zoekrobot je alle pagina's over deze filmsterren voor. Dit komt omdat Google iedere internetpagina op voorhand heeft gelezen en ingedeeld per onderwerp. Maar hoe werkt het dan als je afbeeldingen wil van deze acteurs? In zijn masterscriptie schrijft Ruben Verborgh hoe het intelligente internet Google kan helpen om foto's te vinden.

Momenteel doorloopt Google alle pagina's en verzamelt de afbeeldingen hierop aan de hand van de tekst die erbij staat. Dit is echter niet steeds een goede indicatie voor inhoud van de afbeelding zelf. Als je zoekt op *"Johnny Depp Keira Knightley"*, vind je enkel webpagina's waarop iets over beide celebrities staat. Bij de afbeeldingen zie je echter vanalles, waaronder verbazend genoeg zelfs foto's van Angelina Jolie en Daniel Craig. Blijkbaar helpt de tekst op de pagina Google niet genoeg om de foto's te begrijpen. Google weet alles over teksten, maar zo weinig over afbeeldingen?

Laat ons in gedachten even zelf voor Google spelen: wij zijn de bibliothecaris van het hele internet, en een vrouw aan de balie vraagt ons om een foto van haar twee favoriete filmsterren. Daarom hebben we de afgelopen week – of zeg maar 10 jaar – alle foto's op het hele internet bekeken, genummerd, gezichten herkend en het nummer bij de juiste naam geplaatst in een lange lijst. Een reuzenwerk, maar het lukt, tenminste voor de mensen die we herkennen. Hoe komen we dan te weten wie de vreemde man met baard naast Angelina Jolie is? En hoe heet de jongste dochter van Prins Filip nu weer? Dankzij Wikipedia kunnen we ook Brad Pitt en prinses Eléonore aan de lijst toevoegen.

In zijn masterscriptie aan de Universiteit Gent dacht Ruben Verborgh na over hetzelfde probleem, maar vroeg zich af hoe computers dit langdurige en vervelende werk in onze plaats kunnen uitvoeren. Hij stootte daarbij op enkele belangrijke problemen. Er bestaan wel een hoop programma's die mensen herkennen, maar die werken nog niet echt betrouwbaar. Sinds kort probeert Facebook bijvoorbeeld ook om je vrienden automatisch te *taggen* in je fotoalbum. Met de nadruk op "probeert", want de software heeft het wel vaker bij het verkeerde eind. Gelukkig laat Facebook je zelf kiezen of je de suggesties aanvaardt, zodat je kleine zusje niet per ongeluk aanwezig lijkt op die uitgangsavond met je vrienden. Maar zelfs voor een gigantisch bedrijf als Google is het onmogelijk om genoeg mensen in te schakelen om de dagelijkse stroom aan miljoenen nieuwe afbeeldingen te bekijken.



About: Prince Philippe, Duke of Brabant

http://dbpedia.org/page/Prince_Philippe%2C_Duke_of_Brabant

About: Prince Philippe, Duke of Brabant

An Entity of Type : person, from Named Graph : http://dbpedia.org, within Data Space : dbpedia.org

Prince Philippe, Duke of Brabant, Prince of Belgium, styled His Royal Highness The Duke of Brabant, is the eldest son and heir apparent of Albert II, King of the Belgians and Queen Paola. His godparents were his paternal grandfather, King Leopold III of the Belgians and his maternal grandmother Donna Luisa, Princess Ruffo di Calabria.

Property	Value
dbpedia-owl:abstract	• Philippe Léopold Louis Marie, Prinz von Belgien, Herzog von Brabant aus dem Haus Sachsen-Coburg und Gotha, ist der älteste Sohn und Thronerbe von König Albert II. der Belgier und Königin Paola. Philippe ist sein französischer [illegible] Auf Niederländisch heißt der belgische Kronprinz offiziell Filips Leopold Lo[illegible] Ma[illegible] ber umgangssprachlich wird er meistens Filip genannt. • Prince Philippe, Duke of Brabant, Prince of Belgium, st[illegible] Ro[illegible] ghness The Duke of Brabant, is the eldest son and heir appar[illegible] e Belgians and Queen Paola. His godparents were his paternal grandfather, Ki[illegible] opold III of the Belgians and his maternal grandmother Donna Luisa, P[illegible] di Cal[illegible] • El Príncipe Felipe (Philippe Léopold Louis Marie), nacio [illegible] selas el 15 [illegible] le 1960. [illegible] la Reina [illegible] sorte Paola Ru[illegible] alabria. Es el hijo mayor del Rey [illegible] Sus padrinos fueron el Re[illegible] abuela paterna). Fue edu[illegible] Bélgica, entre 1978 y 198[illegible] Universidad de Oxford, [illegible]

Foto: © Getty Images

Mocht Google slim genoeg zijn om zelf Wikipedia te lezen, dan zou hij die herkenningsprogramma's kunnen bijsturen zodat menselijke hulp overbodig wordt. Als je weet dat Angelina trouwde met Brad en dat prins Filip een dochtertje van 2 jaar opvoedt, lijkt het plots een stuk makkelijker om iedereen op de foto te herkennen. Eén probleem: hoewel Google maandelijks alle woorden uit Wikipedia haalt, begrijpt hij niet wat de artikels betekenen. Als je zoekt op *"koningin Paola"*, zoekt hij gewoon naar pagina's met *"koningin"* en *"Paola"*, maar niet naar *"koningin van België"*. Gelukkig bestaat er ook zoiets als DBpedia, een soort Wikipedia voor computers, waarop alle artikels vertaald staan in computertaal. DBpedia vormt een onderdeel van het semantisch web, het toekomstige intelligente internet dat men ook wel "Web 3.0" noemt, de volgende evolutie van de digitale snelweg.

Dit bracht Ruben, student computerwetenschappen, op het idee om een programma te schrijven dat – net zoals mensen – op het internet kan surfen om namen van mensen aan foto's te koppelen. Daarna is het een koud kunstje voor Google om die foto's tevoorschijn te halen als je zoekt naar deze mensen. Nieuw aan deze techniek is dat niet alleen de informatie uit de afbeelding zelf, maar de informatie uit het hele internet wordt ingeschakeld om zoekopdrachten te vervullen. Hierdoor krijg je betere resultaten dan wanneer enkel de tekst van de bronpagina meespeelt.

Het programma werd getest op verschillende foto's en blijkt aardig te werken, ook wanneer je de gezichten op de foto moeilijk kan herkennen door schaduwen of onscherpte. Net zoals Wikipedia, weet DBpedia vanalles over de meest uiteenlopende onderwerpen en kan daardoor zeer verschillende situaties onderscheiden. De samenwerking tussen het herkenningsprogramma en het intelligente internet maakt computers dus werkelijk een pak slimmer. Daarnaast kan het programma ook informatie halen uit je Facebookprofiel, zodat hij je vrienden kent (en je kleine zusje). De pixels in de foto komen op die manier tot leven voor Google en andere zoekrobotten.

In de nabije toekomst komen er nog interessante uitdagingen: kunnen we het intelligente internet ook gebruiken om bijvoorbeeld dieren en voorwerpen te herkennen? Is het mogelijk om ook gezichten in YouTube-video's of stemmen in iTunes-liedjes te herkennen? Eén ding staat vast: het intelligente internet zal de manier waarop mensen informatie vinden voor altijd veranderen.