Natural Language Processing: Integration of Automatic and Manual Analysis

Natürliche Sprachverarbeitung:
Integration automatischer und manueller Analyse
Zur Erlangung des akademischen Grades Doktor-Ingenieur (Dr.-Ing.)
genehmigte Dissertation von Dipl.-Inform. Richard Eckart de Castilho aus Bensheim
2014 - Fachbereich Informatik — Darmstadt — D 17

TECHNISCHE
UNIVERSITÄT
DARMSTADT

UBIQUITOUS
KNOWLEDGE
PROCESSING

Natural Language Processing:
Integration of Automatic and Manual Analysis
Natürliche Sprachverarbeitung:
Integration automatischer und manueller Analyse

Genehmigte Dissertation von Dipl.-Inform. Richard Eckart de Castilho aus Bensheim

1. Gutachten: Prof. Dr. Iryna Gurevych, Technische Universität Darmstadt
2. Gutachten: Prof. Dr. Andreas Henrich, Otto-Friedrich-Universität Bamberg
3. Gutachten: Prof. Christopher D. Manning, PhD., Stanford University

Tag der Einreichung: 16. Dezember 2013
Tag der Prüfung: 10. Februar 2014

Darmstadt — D 17

To my family.

Erklärung zur Dissertation

Hiermit versichere ich, die vorliegende Dissertation ohne Hilfe Dritter nur mit den angegebenen Quellen und Hilfsmitteln angefertigt zu haben. Alle Stellen, die aus Quellen entnommen wurden, sind als solche kenntlich gemacht. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Darmstadt, den 16. Dezember 2013

(Richard Eckart de Castilho)

Wissenschaftlicher Werdegang des Verfassers[2]

1998–2006	Studium der Informatik Technische Universität Darmstadt
2006	Abschluss als Diplom-Informatiker Diplomarbeit: *A Framework For Storing, Managing and Querying Multi-Layer Annotated Corpora* Technische Universität Darmstadt
2006-2008	Wissenschaftlicher Mitarbeiter am Institut für Sprach- und Literaturwissenschaft Technische Universität Darmstadt
seit 2009	Wissenschaftlicher Mitarbeiter am Ubiquitous Knowledge Processing Lab Technische Universität Darmstadt

[2] Gemäß § 20 Abs. 3 der Promotionsordnung der Technischen Universität Darmstadt.

Zusammenfassung

Es besteht ein aktueller Trend, natürliche Sprachverarbeitung zur Beantwortung von Forschungsfragen aus dem Bereich der Geisteswissenschaften einzusetzen. Dies erfordert eine Integration automatischer und manueller Analyseschritte, z.B. um zunächst eine Theorie zu entwickeln, um diese Theorie dann anhand einer Korpusstudie zu überprüfen, um Trainingsdaten für ein maschinelles Lernverfahren zu generieren, um ein solches zur automatischen Analyse einzusetzen oder um die Qualität der automatischen Analyse auszuwerten. Manuelle Analysen werden meist von Linguisten, Philosophen und Forschern anderer geisteswissenschaftlichen Disziplinen durchgeführt. Automatische Analysen hingegen werden häufig von Forschern mit umfassenden Programmierkenntnissen durchgeführt, z.B. von Informatikern und zunehmend von Computerlinguisten. Es ist wichtig diese unterschiedlichen Forschergruppen, deren Werkzeuge und Daten näher zusammenzubringen, um in kürzerer Zeit und mit weniger Aufwand Ergebnisse von höherer Qualität zu erzielen. Allerdings werden vielversprechende Kooperationen, die sowohl manuelle als auch automatische Analyseschritte umfassen, z.B. die Analyse großer Korpora, derzeit von vielerlei Problemen behindert:

- *Es existiert keine umfassende Sammlung von interoperablen Softwarekomponenten zur automatischen Textanalyse.*
- *Einen automatischen Analyseablauf aus solchen Komponenten zusammenzustellen gestaltet sich zu schwierig.*
- *Werkzeuge zur automatischen Textanalyse, Textexploration, und zur Erstellung von Annotationen sind nicht interoperabel.*
- *Die Portabilität von automatischen Analyseabläufen zwischen verschiedenen Computern ist nicht gegeben.*
- *Arbeitsabläufe sind nicht einfach auf Clusterumgebungen zu übertragen.*
- *Es existieren keine angemessenen Werkzeuge zur selektiven Annotation innerhalb großer Korpora.*
- *Bei der automatischen Analyse sind Annotationstypen vorgegeben, während bei der manuellen Analyse eine flexible Gestaltung der Annotationstypen erforderlich ist.*
- *Die Implementierung neuer Analysekomponenten ist zu aufwändig.*
- *Analysekomponenten und -abläufe können nicht leicht auf Fehler untersucht oder einer Refaktorisierung unterzogen werden.*
- *Analyseabläufe, deren Struktur sich anhand ihrer Parametrisierung dynamisch ändert, werden nicht direkt unterstützt.*
- *Der Benutzer behält keine Kontrolle über automatische Analyseabläufe, wenn zu deren Nutzung fremdes Expertenwissen, undokumentiertes Wissen, oder externe Infrastrukturen, z.B. Webservices, notwendig sind.*

In Zusammenarbeit mit Wissenschaftlern aus den Geisteswissenschaften erarbeiten wir innovative technische Lösungen und Entwürfe, welche die Verwendung automatischer Textanalyse erleichtern und deren Integration in manuelle Analyseverfahren fördern. Dazu bearbeiten wir vier Bereiche, in denen wir jeweils Grundlagen schaffen, um die oben genannten Probleme zu adressieren:

- Wir verbessern die *Benutzbarkeit* automatischer Analysekomponenten und -abläufe, z.B. durch einfachere Programmierschnittstellen und durch einen Selbstkonfigurationsmechanismus für Analysekomponeten.

- Wir adressieren die *Reproduzierbarkeit* von Analyseergebnissen durch Konzepte, welche die Portabilität und Automatisierbarkeit von Analyseabläufen verbessern. Dadurch werden unnötige manuelle Zwischenschritte in Analyseabläufen vermieden und der Austausch von Abläufen und Ergebnissen zwischen Forschern erleichtert.
- Wir schaffen *Flexibilität* durch die Bereitstellung einer umfangreichen Sammlung interoperabler Analysekomponenten. Darüber hinaus untersuchen wir die Annotationstypen unterschiedlicher Komponentensammlungen auf wiederkehrende Entwurfsmuster, die Raum für Anpassungen im Rahmen manueller Analysen bieten.
- Wir schaffen *Interaktivität* im Umgang mit automatisch erstellten Analyseergebnissen, indem wir die Suche über Korpusdaten direkt in einen mehrbenutzerfähigen Annotationsprozess integrieren. Diesen neuartigen Ansatz der *Annotation durch Suche*, unterstützen wir weiterhin durch ein neues webbasiertes Annotationswerkzeug.

Wir demonstrieren die Tragfähigkeit unserer Konzepte anhand von Beispielen, die prototypisch für ganze Klassen von Forschungsproblemen stehen. Zudem haben wir alle vorgestellten Konzepte in bestehende Open-Source-Projekte integriert, oder als neue Open-Source-Projekte umgesetzt und veröffentlicht.

Abstract

There is a current trend to combine natural language analysis with research questions from the humanities. This requires an integration of automatic analysis with manual analysis, e.g. to develop a theory behind the analysis, to test the theory against a corpus, to generate training data for automatic analysis based on machine learning algorithms, and to evaluate the quality of the results from automatic analysis. Manual analysis is traditionally the domain of linguists, philosophers, and researchers from other humanities disciplines, who are often not expert programmers. Automatic analysis, on the other hand, is traditionally done by expert programmers, such as computer scientists and more recently computational linguists. It is important to bring these communities, their tools, and data closer together, to produce analysis of a higher quality with less effort. However, promising cooperations involving manual and automatic analysis, e.g. for the purpose of analyzing a large corpus, are hindered by many problems:

- *No comprehensive set of interoperable automatic analysis components is available.*
- *Assembling automatic analysis components into workflows is too complex.*
- *Automatic analysis tools, exploration tools, and annotation editors are not interoperable.*
- *Workflows are not portable between computers.*
- *Workflows are not easily deployable to a compute cluster.*
- *There are no adequate tools for the selective annotation of large corpora.*
- *In automatic analysis, annotation type systems are predefined, but manual annotation requires customizability.*
- *Implementing new interoperable automatic analysis components is too complex.*
- *Workflows and components are not sufficiently debuggable and refactorable.*
- *Workflows that change dynamically via parametrization are not readily supported.*
- *The user has no control over workflows that rely on expert skills from a different domain, undocumented knowledge, or third-party infrastructures, e.g. web services.*

In cooperation with researchers from the humanities, we develop innovative technical solutions and designs to facilitate the use of automatic analysis and to promote the integration of manual and automatic analysis. To address these issues, we set foundations in four areas:

- *Usability* is improved by reducing the complexity of the APIs for building workflows and creating custom components, improving the handling of resources required by such components, and setting up auto-configuration mechanisms.
- *Reproducibility* is improved through a concept for self-contained, portable analysis components and workflows combined with a declarative modeling approach for dynamic parametrized workflows, that facilitates avoiding unnecessary auxiliary manual steps in automatic workflows.
- *Flexibility* is achieved by providing an extensive collection of interoperable automatic analysis components. We also compare annotation type systems used by different automatic analysis components to locate design patterns that allow for customization when used in manual analysis tasks.
- *Interactivity* is achieved through a novel *annotation-by-query* process combining corpus search with annotation in a multi-user scenario. The process is supported by a web-based tool.

We demonstrate the adequacy of our concepts through examples which represent whole classes of research problems. Additionally, we integrated all our concepts into existing open-source projects, or we implemented and published them within new open-source projects.

Acknowledgements

First and foremost, I would like to thank Prof. Dr. Iryna Gurevych for giving me the opportunity to conduct my research, for her valuable guidance, feedback, and advice. I would also like to thank my secondary supervisors Prof. Dr. Andreas Henrich and Prof. Chris Manning for taking the time to evaluate my thesis. Parts of this research were supported by the German Federal Ministry of Education and Research (BMBF) under the promotional reference 01UG1110D, by the German Federal Ministry of Economy and Technology under the promotional reference 01MQ07012, and by the Hessian research excellence program "Landes-Offensive zur Entwicklung Wissenschaftlich-ökonomischer Exzellenz" (LOEWE) as part of the research center "Digital Humanities".

I would like to thank my current and former colleagues at the Ubiquitous Knowledge Processing Lab for countless discussions and for their support. Special thanks go to Dr. Torsten Zesch and Dr. Judith Eckle-Kohler, who always generously offered me their time, advice, and at times their critique. I would like to thank the DKPro Core crowd, in particular Nicolai Erbs, Pedro Santos, and all others at the UKP Lab and outside who used DKPro Core and who provided feedback on it. I would like to thank the early adopters of DKPro Lab, in particular Benjamin Herbert, Oliver Ferschke, and Johannes Daxenberger. Their feedback was essential to the success of the project. Thanks also go to the LOEWE-NCC team, Erik-Lân Do Dinh, Dr. Sabine Bartsch, Pia Weber, Dr. Janina Radó, and Prof. Dr. Gert Webelhuth, for the great collaboration and the interesting and productive discussions. Finally, thanks go to the WebAnno team in particular to Prof. Dr. Chris Biemann, Seid Muhie Yimam, and to the other CLARIN FAG-7 members.

Furthermore, I would like to extend my gratitude to the communities of uimaFIT, of the Apache UIMA framework, and of the Stanford CoreNLP framework, in particular Philip Ogren, Steven Bethard, Marshall Schor, and John Bauer. Without their good will and valuable feedback, my work would not have been possible.

Additional thanks go to Peter Klügl and Dr. Katrin Tomanek for having me on the organization teams of the 2nd and 3rd UIMA@GSCL workshops in 2009 and 2013.

As my way up to this point was long and twisted, I am grateful to these additional people who helped me and prodded me at various steep passages and crossroads: Prof. Dr. Elke Teich, Dr. Mônica Holtz, Dr. Georg Rehm, Prof. Dr. Christian Chiarcos, Prof. Nancy Ide, Anthony Levas, and Paul Keyser.

Finally, I would like to thank my family and friends for their love and their support. This endeavor would surely not have been possible without them!

Contents

1 Introduction

1.1 Motivation

There is a current trend to combine natural language analysis with research questions from the humanities. This requires an integration of automatic analysis with manual analysis, e.g. to develop a theory behind the analysis, to test the theory, to generate training data for automatic analysis based on machine learning algorithms, or to evaluate the quality of automatic analysis results. Manual analysis is traditionally the domain of linguists, philosophers, and researchers from other humanities disciplines, who are often not expert programmers. Automatic analysis, on the other hand, is traditionally done by expert programmers, such as computer scientists and more recently computational linguists.

So far, this had the effect that many automatic analysis could not be used effectively by non-expert programmers, due to the complexity of setting them up and combining them into complex analysis workflows. Furthermore, communities doing manual and automatic analysis have different requirements towards tools and often rely on different tool stacks, making it difficult to set up a working environment which integrates automatic as well as manual analysis.

It is important to bring these communities, their tools, and their data closer together, to produce analysis of a higher quality with less effort. Consider the following scenario in which a linguist, and a computer scientist decided to collaborate. It illustrates prototypically the interaction between manual and automatic analysis (Figure 1.1):

- The linguist is doing fundamental research, comparing the use of certain uncommon grammatical constructions in different languages. To find a sufficient number of such constructions in real-world text, he needs to examine a large corpus for each language, larger than he can examine without any automated assistance.
- The linguist believes to have a basic idea of how to find the constructions he is looking for, based on patterns over linguistic categories. If all of his corpora were already annotated with these linguistic categories, he may be able to find what he is looking for more easily. He manages to find several analysis tools that can annotate the desired categories, but these ❶ **analysis tools are not interoperable**. After investing a considerable amount of time ❷ **building a makeshift script** binding them together, he gets some analysis results suitable for an initial exploration for a subset of his data, as a full processing was not possible on his workstation. There is ❸ **no annotation editor into which he can read his analysis results**, so he starts using a spreadsheet to make some records.

Definition: ***analysis workflows*** – A set of analysis components that are applied to primary data in a certain order to produce an analysis result.

Definition: ***analysis tools*** – A standalone software for language analysis, i.e. the software is not integrated with a particular processing framework.

Definition: ***annotation editor*** – An application which allows conducting a manual analysis of language data by inspecting and editing annotations over primary data, such as text.

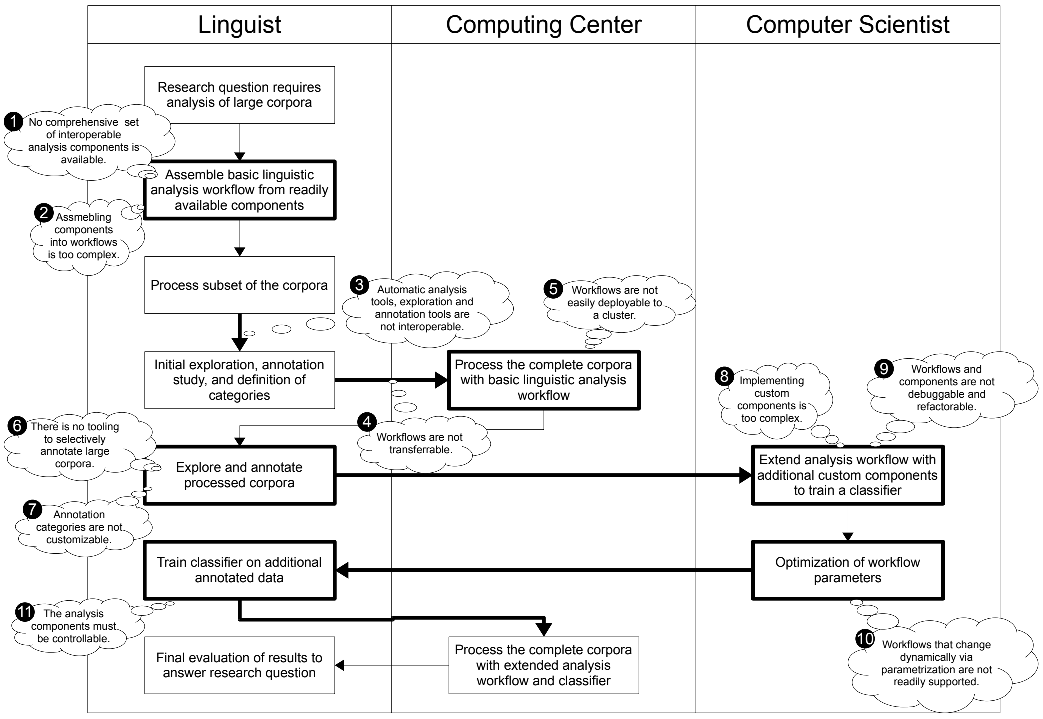

Linguist
Computing Center
Computer Scientist
Research question requires analysis of large corpora
1 No comprehensive set of interoperable analysis components is available.
Assemble basic linguistic analysis workflow from readily available components
2 Assmebling components into workflows is too complex.
Process subset of the corpora
3 Automatic analysis tools, exploration and annotation tools are not interoperable.
5 Workflows are not easily deployable to a cluster.
Initial exploration, annotation study, and definition of categories
Process the complete corpora with basic linguistic analysis workflow
8 Implementing custom components is too complex.
9 Workflows and components are not debuggable and refactorable.
6 There is no tooling to selectively annotate large corpora.
4 Workflows are not transferrable.
Explore and annotate processed corpora
Extend analysis workflow with additional custom components to train a classifier
7 Annotation categories are not customizable.
Train classifier on additional annotated data
Optimization of workflow parameters
11 The analysis components must be controllable.
10 Workflows that change dynamically via parametrization are not readily supported.
Final evaluation of results to answer research question
Process the complete corpora with extended analysis workflow and classifier

Figure 1.1: Use case: a linguist and a computer scientist collaborate on analyzing a large corpus

- To get the complete corpora processed, the linguist sends his script to the operator of a computing center. The operator is supposed to run the workflow on a compute cluster. The linguist's ❹ **script does not run on the cluster**. The operator has to go through various phone calls with the linguist to work out ❺ **what tools in which version and with which resources** the script uses. Eventually, the operator manages to install all of these on the cluster, rewrites the script for the cluster, processes the data, and sends it back to the linguist.
- The linguist imports the processed corpora into a linguistic search engine and starts looking for his constructions. He expresses his intuition about his grammatical constructions as query patterns. With some specialized queries, the linguist achieves an adequate precision identifying the constructions he is interested in. However, as he starts relaxing the conditions in his queries, he starts getting many query results that are not the constructions he is looking for. Since ❻ **the search engine does not support categorizing search results**, he records those constructions he has found in his corpora and those that were wrongly returned by the queries in a spreadsheet. At least with a spreadsheet he is not ❼ **limited to an annotation structure prescribed by some tool**. He sends this spreadsheet and his script to the computer scientist, for her to train a classifier.
- The computer scientist goes through the same routine of phone calls to figure out which tools and resources to install as the operator before. Then, she reimplements the makeshift script in another programming language, because she needs to ❽ **integrate additional analysis components** of her own in a way that she can easily ❾ **debug the setup and has short turn-around cycles for bug-fixing**. She preprocesses the training data from the spreadsheet, and uses another quickly written script to extract features to train the

Definition: ***resources*** – Many analysis tools require resources, such as statistical or probabilistic models, dictionaries, word lists, etc.

classifier. After several ❿ **modifications of the analysis workflow**, a parametrization of the components has been found that produces good results (the notebook page containing the discarded parametrizations and results is lost soon after). She cannot send her training and classification workflows back to the linguist, as he does not have the necessary development environment installed to run them (much less an idea of how to do that).

- The linguist has meanwhile produced additional training data. Instead of training a classifier using the training workflow from the computer scientist, he has to send his new data to the computer scientist. He feels uncomfortable about this, because once the cooperation is over, he cannot use the developed approach to train classifiers for new data, since he has ⓫ **no control over the analysis components** used in the workflow. Meanwhile, spreadsheets and classification results are exchanged in several iterations between the linguist and the computer scientist, gradually improving the classifier performance.
- Eventually, it is decided that the classifier works well enough and that it needs to be applied to the full corpora. The extended analysis workflow and the custom components are sent to the computing center operator to analyze the full corpora on the cluster. This time, the operator goes through several iterations with the computer scientist figuring out how to integrate the custom components into the script he runs on the cluster. In the end, the operator sends the processed and classified data back to the linguist.
- Now, finally, the linguist has enough data to conduct his research on the uncommon grammatical structures.

1.2 Requirements

It is the goal of the integration of manual and automatic analysis to build applications and to define best practices which account for both modes of analysis, to make automatic analysis available to a larger clientele and to bring the parallel worlds of data formats and tool chains closer together. Existing work addresses particular aspects of an integrated scenario such as is illustrated in Section 1.1, but none addresses it completely. The necessity to take into account all requirements from the big picture (Figure 1.1 on page 2) makes it particularly difficult to find viable solutions for such a scenario. There are multiple issues which need to be addressed in order to integrate manual and automatic analysis:

❶ No comprehensive set of interoperable automatic analysis components is available.
❷ Assembling automatic analysis components into workflows is too complex.
❸ Automatic analysis tools and annotation editors are not interoperable.
❹ Workflows are not portable between computers.
❺ Workflows are not easily deployable to a compute cluster.
❻ There are no adequate tools for the selective annotation of large corpora.
❼ In automatic analysis, annotation type systems are predefined, but manual annotation requires customizability.
❽ Implementing new interoperable automatic analysis components is too complex.

Definition: *analysis components* – A software tool for language analysis which has been wrapped as a reusable component for a processing framework.

Definition: *annotation type systems* – A set of annotation types which often interact, e.g. one type bears a feature whose value is an annotation of another type.

❾ Workflows and components are not sufficiently debuggable and refactorable.
❿ Workflows that change dynamically via parametrization are not readily supported.
⓫ The user has no control over workflows that rely on expert skills from a different domain, undocumented knowledge, or third-party infrastructures, e.g. web services.

As we examine the state-of-the-art in processing frameworks for natural language analysis, we find these issues to be insufficiently addressed.

1.3 Contributions

In this section we briefly present our contributions to address the requirements. In cooperation with researchers from the humanities, we developed innovative technical solutions and designs to facilitate the use of automatic analysis and to promote the integration of manual and automatic analysis. We grouped our contributions into four areas, which serve as our guiding principles: *usability*, *reproducibility*, *flexibility*, and *interactivity*. The black-circled numbers behind each contribution refer back to the issues identified in the illustrative use case, which are addressed by our proposed solutions.

Usability

Assembling automatic analysis workflows has to be convenient, for non-expert programmers, such as the linguist, as well as for expert programmers, such as the computer scientist. Existing software for automatic language analysis tends to be either very powerful and configurable, making it attractive for the computer scientist, or easy to use, making it attractive for the linguist. When embedded within tools with convenient graphical user interfaces, automatic analysis is embraced by linguists and other non-expert programmers. The implementation of analysis workflows or analysis components often requires a significant amount of boiler plate code – simple APIs targeted towards non-expert programmers are rarely available. Therefore, a way to facilitate the integration of automatic analysis workflows into the processes of humanities researchers is to simplify their assembly and deployment of the workflows.

In addition, it is inconvenient to manually install additional analysis tools on the local system before they can be used. Given a description of an analysis workflow, the processing framework should automatically acquire and install the necessary analysis components and resources.

While analysis components should also be configurable, it should not be mandatory to explicitly provide a configuration. Analysis components should be usable out-of-the-box.

To improve the usability for non-expert programmers while retaining all flexibility for expert programmers, we propose these measures:

1. An **auto-configuration mechanism** which allows an analysis component to automatically determine the value of a configuration parameter depending on the context it is running in and on the primary data it is processing. For an optimal effect, we combine this with a standardized addressing and packaging scheme for resources, such as parser model files,

Definition: *processing framework* – A piece of software which enables the interoperability of analysis components and facilitates their use. The framework defines a life cycle for analysis components, means of configuring analysis components, as well as a common data structure which all analysis components use to exchange data with each other. Beyond this, a processing framework may offer many additional features, e.g. different workflow strategies, the ability to scale the processing to large data and over multiple computers, a serialization format for analyzed data, etc.

allowing the automatic acquisition of the required resources from a repository. Through overrides, the mechanism still allows full customization of the components. This way, it is possible to eliminate the need for explicit configuration for many analysis components. ❷ ❺ → Section 3.1

2. An **improved API** for analysis components and workflows with a particular focus on the configuration of components. Processing frameworks allow the composition of components into analysis workflows. For the components to be reusable effectively, they must be configurable, so that they can be contextualized when a workflow is assembled for a particular kind of analysis. We extended the ability of a processing framework to allow the extraction of key behavioral aspects of the analysis component into a pluggable object, thus making the component's behavior customizable. ❷ ❽ → Section 3.2

Reproducibility

Reliable, repeatable analysis workflows are essential for reproducible research, but manual steps are inherently error prone and often badly documented. Automatic workflows tend to be interspersed with unnecessary, auxiliary manual steps, because natural language processing frameworks provide little or no out-of-the-box support for dynamic workflows. Also, while some tools support the sharing of workflows, these are not self-contained, requiring the user to manually ensure that all used components and resources are present in the correct versions. In this case, it is necessary to *remove* the haphazard integration of manual intervention into otherwise automatic analysis.

Researchers should be supported by tools as well as by best practice guidelines in creating reproducible automatic workflows. When a workflow is applied to the same data, it should reproducibly deliver the same results. When applied to similar data, it should reliably produce similar results.

Research is an inherently repetitive process involving multiple iterations over the same problem or the same data to get reliable results. Tools to support reproducible experiments should therefore also support flexible means of parameter variation.

A workflow setup should be self-describing and explicitly declare which specific versions of analysis components, resources, and primary data it uses. These should be under the full control of the researcher, e.g. not restricted by certain licenses or hidden behind ephemeral web services, which may change incompatibly or even disappear any time.

To facilitate fully automatic analysis workflows, this thesis provides:

3. An **approach for reproducible analysis workflows** based on components and resources, packaged, versioned, discoverable, and distributable via a reliable infrastructure to enable materializing them easily on a workstation or cluster system and to reuse already materialized resources between workflow executions to reduce turn-around times. ❹ ❺ ⓫ → Section 4.1
4. A **declarative approach for modeling dynamic, parametrized workflows**, allowing for a clean decomposition of the experiment into the space of parameters to be explored, the

Definition: ***primary data*** – The data being subject to analysis, e.g. text documents. In an annotated corpus, the primary data is only the text without any annotations.

Definition: ***repository*** – A repository is a central place used to archive and share the components and resources used by an analysis workflow, their dependencies, and possibly the workflow itself.

analysis tasks to be performed for each parameter configuration, and the dependencies between the tasks. ❾ ❿ → Section 4.2

Flexibility

Each user has individual goals and requirements. Being able to flexibly configure and combine analysis tools into workflows is key to gaining acceptance in a larger user community. For manual analysis, customization is essential, particularly during early exploratory annotation phases where annotation guidelines are constantly subject to change, so that there are no fixed annotation types or tag sets. However, for automatic linguistic analysis components, annotation types and tag sets are a part of the interface specification between the components and key to enabling interoperability.

The data models used by analysis components for interoperability should be developed conscious of the flexibility required during phases of exploration and preliminary analysis. The requirement for maximal flexibility as posed in particular by the preliminary analysis contradicts the requirement for well-known types which allow automatic analysis components to interoperate. For example, if the annotation type used to represent part-of-speech tags restricts these to a certain tag set, a researcher working with an annotation editor which is using that data model cannot define new tags during an exploratory phase without changing the data model.

Analysis components should be flexibly combinable into analysis workflows. Components performing the same kind of analysis but using different algorithms or implementations should be usable interchangeably within such workflows. Therefore, it is important to make comprehensive collections of interchangeable and interoperable analysis components readily available.

Towards this, we take the following steps:

5. A **systematic comparison of different annotation type systems** that enable interoperability between automatic analysis components allows us to identify design patterns that provide room for customization. This provides us with a better understanding of how to balance customization against interoperability in an integrated scenario. ❸ ❼ → Section 5.1
6. A **collection of analysis components** has been considerably extended, unified, and refactored for interoperability, and interchangeability. In our example scenario, this contributes significantly to the ability of the non-expert programmer to flexibly combine components into a custom analysis workflow. We discuss how interoperability and interchangeability has been achieved and the decisions that have been taken in the process. ❶ → Section 5.2

Interactivity

Manual and automatic analysis should interact with each other. The manual analysis of large corpora is not feasible without automated assistance. Likewise, it is difficult to reach high-quality annotations and to measure quality in terms of inter-annotator agreement, without multiple annotators working collaboratively. It is necessary to define a process which integrates automatic analysis and a collaborating team of human annotators to enable the analysis of large corpora.

Definition: *annotation types* – A type to distinguish between different kinds of annotations, bearing different attributes and semantics. For example, when an annotation is realized as feature structure, the type defines which features can be used. A feature structure of the type *PartOfSpeech* may provide a feature *posTag*.

Automatic analysis can be used to support the exploration and manual analysis. For example, a classifier can be trained using several examples of a particularly interesting grammatical construction and could be used to locate those constructions in a large corpus. Such an approach allows a user to locate potential occurrences of the construction, without having to manually examine the entire data set.

The manual correction of automatically analyzed data can be used to improve the subsequently performed automatic analysis, either by retraining a model for an analysis component or by passing the corrected annotations to analysis components being executed later in the workflow.

The system should provide information about the quality of the analysis, for example by reporting on the agreement between different users analyzing the same data. A system could try to learn models from manually created or corrected annotations using different sets of features and report to the user which features provide the highest precision or the highest accuracy.

Based on the above suggested use-case of annotating infrequent grammatical constructions in large corpora, we propose a process integrating automatic and manual analysis and provide a tool supporting it:

7. The **annotation-by-query** process combines the search in a corpus with annotation. A large corpus is first automatically analyzed and annotated for relevant categories. Queries over these annotations are then formulated based on a research hypothesis. The query results are annotated as *correct* if they are true occurrences of a desired phenomenon, and otherwise as *wrong*. Statistical evaluation of these annotations lets the researcher improve the queries and the underlying hypothesis iteratively. The query-based approach requires the introduction of new kinds of interaction between annotators in order to assure annotators are not working on completely separate parts of the corpus and to maintain the ability of calculating annotator agreement. At the same time, annotators remain isolated from each other to avoid bias. The *annotation-by-query* process is further supplemented by machine learning. Once some results have been marked, a classifier can be trained on the pre-annotated data and the *correct/wrong* labels. When a query has many results, the classifier can help the annotator to focus on results with a higher probability of being *correct*. ❻ → Section 6.1

1.4 Challenges

The challenge we had to face in developing our contributions was to balance the guiding principles within each contribution. E.g. contributions towards reproducibility should not incur a loss of usability. Below, we address pairs of guiding principles, which were particularly difficult to balance.

Usability vs. reproducibility

The rigid and detailed control required to create a reproducible analysis workflow needs to be mitigated through the use of mechanisms and infrastructures that allow enforcing this control without much user effort. While such mechanisms and infrastructures exist, they have not been embraced by current state-of-the-art processing frameworks such as GATE [51], Tesla [194] or UIMA [83]. To address this, we propose an approach to building self-contained, portable analysis workflows (Section 4.1) building on a public, reliable, controllable, and convenient distribution infrastructure. To incorporate language resources, such as parser models, into this approach, we define packaging conventions for deploying resources to this infrastructure (Section 3.1). Finally, our collection of analysis components, DKPro Core (Section 5.2), has been extended to provide such packaged models and it is provided via now via the distribution infrastructure. Hence, it is now possible to build portable analysis workflows using our components.

Reproducibility vs. interactivity

Manually performed tasks are sources of uncertainties and mistakes, which inherently conflicts with the goal of reproducibility. In a manual analysis setup, a well-designed annotation task structure, annotation guidelines, and a sufficiently large team of annotators can mitigate such problems, but not remove them. In an automatic analysis setup, it is important that no auxiliary or intermediate steps remain, which need to be performed manually.

However, there are often setups which could be done fully automatically, but certain steps are not automatized because there is no convenient way to integrate them into the workflow mechanism of the processing framework being used. In our experience, these are often preprocessing steps that need to be seldomly re-run, the reconfiguration of steps to re-run them with different parameters, or steps aggregating results from runs with different parameters. Moreover, these steps are often badly documented, if at all. For a truly reproducible setup, it is necessary to eliminate such manual steps and integrate them into the automatically executable workflow. To facilitate this, we designed a lightweight framework for parameter-sweeping experiments ([72], Section 4.2), which allows implementing complex analysis workflows independent of any specific processing framework in a declarative manner with minimal overhead.

Flexibility vs. usability

The complexity caused by powerful configuration capabilities needs to be countered. We suggest an auto-configuration mechanism (Section 3.1) that sets appropriate parameter values depending on the analysis setup or on the data being processed. To the best of our knowledge, such mechanisms are not yet available in current state-of-the-art language processing frameworks. We integrated this mechanism into our collection of analysis components, DKPro Core (Section 5.2), thereby improving its usability.

We also address the configuration of analysis components itself. It is often desirable to configure analysis components with user-controlled custom strategies that have a significant influence on the behavior of the component, e.g. to extract features for machine learning, matching, ranking, or just to make certain processing aspects highly configurable. However, doing this is not trivial, as the configuration capabilities of processing frameworks focus mainly on composing analysis workflows from analysis components and not on the composition of arbitrary systems, e.g. by restricting configuration parameters to certain data types.

We propose a generalization of the configuration capabilities to facilitate the use of custom strategies (Section 3.2). Our approach operates on the level of a general purpose programming language (e.g. Java or Groovy) instead of implementing a domain specific language like JAPE (*Java Annotation Patterns Engine*) [52] or *Ruta* (*Rule-based Text Annotation*) [133]. Popular general purpose languages enjoy extensive support from integrated development environments with respect to debugging and refactoring, and we aim to benefit from this. However, we design an API which mimics the convenience of a domain specific language and aims to be usable by non-expert programmers.

Interactivity vs. flexibility

To provide interoperability between analysis components and to enable the flexibility of using similar components interchangeably in a workflow, a well-defined model for the exchanged data is required. However, this model also needs to provide degrees of freedom, so it can be used during preliminary analysis while the analysis guidelines and categories are still subject to frequent changes. We analyze several annotation type systems (Section 5.1) with respect to their design and potential for customization to gain an understanding if and how an annotation type system can be designed to support both, manual and automatic analysis tasks.

1.5 Publication record

Parts of this thesis have been already been published at peer-reviewed conferences and workshops. Implementations of the concepts presented here have largely been published as new open-source software projects or contributed to existing open-source projects.

We describe an approach to the dynamic selection and acquisition of resources required by analysis components for the execution of an analysis workflow (Section 3.1). This approach has been integrated into DKPro Core (see Section 5.2, [62]), our open-source collection of analysis components for the Apache UIMA [10] framework.

We present an approach to configuring analysis components with parameters representing complex objects, based on the strategy design pattern (Section 3.2). We integrated the approach into Apache uimaFIT [14], an open-source project providing a simplified API for the Apache UIMA [10] framework.[1] The approach is used by analysis components from the DKPro-Core component collection (see Section 5.2, [62]), as well as by the DKPro Text Classification framework (DKPro TC, [65]).

We describe an approach to portable, reproducible analysis workflows (Section 4.1). This approach is used to provide an easy and convenient way for new users to experiment with DKPro Core (Section 5.2). Additionally, it is used to provide scripts to convert corpora into the data formats required by the CSniper annotation tool (see Section 6.1, [50; 75]).

We describe an approach to analysis workflows that change their structure based on their parametrization, e.g. in a parameter sweeping experiment, which runs the same workflow with many parameter combinations (Section 4.2). Initially, we developed this approach in the context of an experiment on retrieving service descriptions using natural language queries and semantic similarity [73] within the THESEUS TEXO project. Later, we presented it as a generic solution for information retrieval experiments [72]. We made an implementation of our concept available in the new open-source project DKPro Lab [63]. This implementation has since been used by other members of the Ubiquitous Knowledge Processing Lab for a variety of different tasks including the prediction of quality flaws in Wikipedia [85], automatically classifying edit categories in Wikipedia revisions [56], age and gender author profiling in social media [87], preposition and determiner correction [234], and textual entailment [236]. It is also used in the open-source projects DKPro TC [65] and DKPro Spelling [64]. DKPro Spelling is a collection of software components for spelling correction.

We analyze the annotation type systems from several analysis component collections based on the Apache UIMA [10] framework (ClearTK [172], cTAKES [190], DKPro Core [Section 5.2], JCoRe [109], and U-Compare [129]) to identify patterns underlying the design of annotation type systems and analyze their relation to each other (Section 5.1). This analysis serves as a preparatory study for a new extension of the open-source annotation editor WebAnno [232], allowing for the definition of custom annotation types. It is also intended to serve as input for future discussions on a common annotation type system with the communities of the respective component collections.

[1] Apache uimaFIT was initially created under the name *uutuc* and primarily maintained by Philip Ogren and Steven Bethard in 2009. Some time later, I joined the team, contributing initial extensions to handle UIMA shared resources. The project was then renamed to uimaFIT in 2010. In 2013, uimaFIT was donated to the Apache Foundation as part of the Apache UIMA project. Its first release under the new Apache brand was version 2.0.0 in August 2013. At the time of writing, I continue to maintain uimaFIT as a committer in the Apache UIMA project.

Definition: ***analysis component collections*** – A set of analysis components which are immediately interoperable without requiring any kind of data conversion.

We describe extensions to the open-source *DKPro Core*[2] component collection (Section 5.2, [62]). DKPro Core is an essential foundation for most of the research being done at the Ubiquitous Knowledge Processing Lab. It is also used externally, e.g. by the EXCITEMENT Open Platform for textual entailment [78; 167], by Riedl and Biemann [185] in an experiment on text segmentation with topic models, in the JoBimText project[3] [125; 102], and by Strötgen and Gertz [207] in experiments on temporal tagging. We have also presented DKPro Core as part of a tutorial held in conjunction with the 3rd UIMA@GSCL workshop at the GSCL 2013 conference.[4]

We describe a process which integrates linguistic search and annotation over large corpora to locate and annotate infrequent phenomena in large, pre-annotated corpora (Section 6.1). An early version of this approach was presented at the Lingustic Processing Pipelines Workshop at the GSCL 2009 [74]. The concept was further developed [75] in the context of the LOEWE Research Center "Digital Humanities" to conduct a contrastive comparison of non-canonical grammatical constructions between English and German [182]. It was implemented and published in the open-source tool CSniper [50].

[2] DKPro Core is an open-source software which was created in the context of the *Darmstadt Knowledge Repository* (DKPro) - an initiative for creating reusable software components for language analysis at the Ubiquitous Knowledge Processing Lab. I joined the group in 2009 as technical lead with the task of refactoring DKPro and improving the maintainability of the code base. My first release in 2009 was a subset of DKPro Core called DKPro UGD (User-generated discourse). The software was still relying on XML descriptors and was difficult to use. Because of this usability problem, I switched the general strategy to building workflows programmatically using uutuc (today Apache uimaFIT), joined the uutuc development team in 2009, and joined the Apache UIMA development team in 2012. Additionally, the build process was completely migrated to Apache Maven [180] to further streamline the development process. The next release was DKPro Core 1.1.0 in 2011. It was already based on the new programmatic approach. Since then, the set of components in DKPro Core has steadily grown and the usability has been further improved with every new version. The current version of DKPro Core is 1.5.0. Many of the improvements that I have contributed to DKPro Core over the time are described in this thesis. The contributions discussed in this thesis were my own work, whereas other changes, in particular some of the new components that were added to DKPro Core during the time, were contributed by my colleagues at the UKP Lab.

[3] According to a code search engine: `http://code.ohloh.net/search?s=%22de.tudarmstadt.ukp.dkpro.core%22` (Last accessed: 2013-12-11)

[4] 3rd UIMA@GSCL workshop: `http://uima.apache.org/gscl13.html` (Last accessed: 2013-12-15)

2 State of the art

In this chapter, we give a high-level introduction to the state of the art in automatic and manual language analysis. In the following chapters, we provide additional state of the art sections which focus in particular on those aspects relevant to the contribution in question. These additional sections may refer to further work which is specifically relevant to the contribution.

Linguistic analysis is the process of understanding a language sample from a linguistic perspective. Linguistics understands language as a complex system of communication, consisting of several interacting sub-systems, such as:

- **Morphology** – The study of words, their composition from atomic semantic units (*morphemes*) and their inflection for number, gender, case, tense, or other linguistic categories. Compound words and derivations are studied under the heading of *lexical morphology*.
- **Syntax** – The study of the composition of sentences from words and phrases. When syntactic rules extend into the morphology of words, we speak of *morphosyntax*.
- **Semantics** – The study of meaning on any level from morphemes to words or sentences, and even across sentence boundaries.
- **Pragmatics** – The study of what is meant to be accomplished through an act of communication. An advertisement, for example, aims to motivate a potential client to buy the advertised product or idea.

Although it may not be immediately obvious to the average native speaker of a language, these sub-systems are *layered*, largely one building up on the other. For example, it is difficult to understand the pragmatics of an utterance unless the words used, their inflections, grammatical relations, and semantics are understood. Hence, in any kind of linguistic analysis processes, the lower layers are usually analyzed before the higher layers. This is particularly true for systems that perform such an analysis automatically, as these, unlike humans, cannot rely on non-explicit *linguistic competence* [47] which would allow them to skip the explicit analysis of the lower layers.

The detailed analysis of language samples is essential in various disciplines such as linguistics, philology, history, or philosophy. Written texts have been created throughout the centuries and teach us knowledge about culture, science, and economy. Written letters, or today emails, or social media are used to communicate with family, friends, and business partners.

With the ever increasing ubiquity of technical systems that humans work with, the desire to communicate naturally with a computer or with a smart phone in human language is also growing. Modern smart phones can be controlled by speech and give turn-by-turn directions, e.g. to the next pharmacy. Computers no longer only play chess, but beat their human opponents even in knowledge games such as *Jeopardy!*[1] [82]. Therefore, improving the capabilities of computers to automatically analyze language becomes increasingly important.

For the purpose of this work, we assume that language samples exist in the form of written text. Spoken language or transcribed speech pose research questions in their own right, which

[1] In *Jeopardy!*, players are presented with a partial fact from a known domain and have to answer with the missing part in form of a question. For example, in the domain of *Celebrities*: Q: *He is an amphibian who hosts a popular variety show.* - A: *Who is Kermit the Frog?*. *Jeopardy!* is a registered trademark of Jeopardy Productions Inc. in the United States, other countries, or both.

are not treated here. While some of the data models, tools, and applications discussed later may also be applicable to audio or video recordings or even multi-modal documents, we focus on their use for text documents.

2.1 Manual analysis

Technical devices with human language processing or generation capabilities would not have been possible today without the legions of researchers that manually analyzed human language for centuries. And still, language processing technology is far from being perfect and many human languages are still hardly understood or documented. Automatic analysis tools only can be used meaningfully after a basic theory about a language has already been set up and sometimes in order to develop such a theory, but they are hardly meaningful without any underlying theory at all. Even then, the results produced by automatic analysis tools need to be manually reviewed to determine their quality. Therefore, manual analysis is still the principal tool to improve our understanding of language.

Manual analysis has often been done with pen and paper, but today it is increasingly being supported by software. The analysis involves repeatedly marking a part of a text and assigning linguistic information to it, for example marking all the words in a sentence and assigning a part-of-speech tag to each one. This process is also called *annotation* and the markings in the text are called *annotations*. In early stages of an analysis, categories may not have been properly defined yet, so the linguist appreciates the freedom provided by a pen-and-paper approach, like scribbling notes and free text comments, which would not be trivially processable by a computer. Annotation software often requires the inventory of categories to be defined before it can be used for annotation. This kind of software also usually only supports certain kinds of annotations, e.g. marking spans of text and drawing arcs between such spans in contrast to arbitrary scribbles, free form drawings, etc. Based on several iterations of preliminary analysis and discussions with peers, eventually annotation guidelines are produced, which document the categories, how they should be applied and how to deal with borderline cases.

Following the creation of annotation guidelines, larger bodies of text can be analyzed. While the preliminary analysis and the creation of annotation guidelines are mainly driven by experts (e.g. researchers or graduate students), at this stage, less skilled personnel (e.g. undergraduate students) can be employed. The quality of the analysis produced can be measured via the agreement between the different annotators. A low agreement may be countered by improving the annotation guidelines. With respect to the precision of an automatic analysis, the inter-annotator agreement can be seen as an upper limit – even a perfect computer cannot get cases *correct* on which humans do not agree what *correct* actually is.

The preliminary analysis and the guideline-driven analysis are iterative processes, in which the linguistic conceptualization and the annotation guidelines are continually revised and improved.

2.1.1 Roles

Mostly following Dipper et al. [60], we distinguish four roles in a manual analysis task (refer to Chapter 6 for a more detailed description):

- **Explorer** – This role performs preliminary exploration of the corpus data, either to generate linguistic hypotheses or to corroborate already existing hypotheses.

- **Guideline author** – This role defines the categories and the annotation guidelines to be used. It requires expert domain skills, e.g. in the linguistic domain.

- **Annotator** – The role performs manual analysis based on the annotation guidelines. Basic domain skills are required to properly interpret the annotation guidelines.
- **Curator** – This role critically examines the annotations made by the annotators to resolve cases where the *annotators* did not agree.

These roles are supported mainly by two kinds of annotation tools: annotation editors and linguistic search engines.

2.1.2 Annotation editors

Annotation editors mainly support the *annotator* role. The basic functionalities an annotation editor needs to support are: the loading of a text document, the ability to create annotations on the text, and to save the annotated document.

Even in a single-user scenario, it is useful for the annotation editor to support the *guideline author* role by allowing the definition of a controlled vocabulary used for annotation or even to manage structured annotation guidelines within the tool. This becomes even more important when multiple users plan to work collaboratively analyzing a corpus. At this point, an annotation editor should also support a centralized data storage.

When multiple users are working together, there are two modes of operation:

- **Collaborative analysis** – In a collaborative scenario, all users operate on a shared set of annotations. One user can make changes to the annotations created by another user (cf. [204]). Explicit support for the *curator* role is not required in this scenario, because effectively every user annotator is curating the results of any other annotator. There is no aggregation of analysis results, as there is only a single set of annotations. For this reason, there is no possibility to measure the quality of the annotation via inter-annotator agreement. Hence, this approach appears to be suitable at most for preliminary exploratory annotation studies or for interactively testing and developing annotation guidelines.
- **Distributed analysis** – In a distributed scenario, every user works on a separate set of annotations. One user is usually not able to modify or even to see the annotations made by other users. This avoids bias (cf. [232]) and makes it possible to measure inter-annotator agreement. A tool may support the *curator* role and allow a certain user to revise and manually merge analysis results into a final result. Alternatively, automatic strategies can be used to aggregate all agreed-upon analysis results produced into a single final result. Bayerl and Paul [16] mention a *majority method* by which agreement is declared if a certain proportion of the users have produced an identical analysis result at a certain decision point, and a *consensus method* by which agreement is declared only if all users have produced an identical analysis result.

To facilitate annotation or to bootstrap the manual annotation process, some annotation editors include basic automatic preprocessing functionality. For example, the *UAM Corpus Tool* [168] integrates a tokenizer to automatically create generic *segment* annotations which are further classified by the annotator later. It also includes an *autocoding* feature which allows automatically creating annotations based on patterns over already existing annotations and text.

Definition: ***annotation tools*** – A general term for tools allowing users to create and interact with annotations, such as annotation editors, annotation visualizers, or linguistic search engines.

Truly sophisticated automatic analysis is usually not found in an annotation editor. Rather, data is preprocessed externally and later imported into the editor for exploration or correction.

Good annotation editors (e.g. [204; 168]) also offer the capability of searching over the annotated text or sometimes even over the annotations. However, these search functions are much less sophisticated than those offered by dedicated linguistic search engines (e.g. [77; 233; 187]), because the latter generally assume a static data set and thus have more options of building efficient index structures.

2.1.3 Linguistic search engines

A linguistic search engine supports the *explorer* role. It provides a means of searching over large annotated corpora using complex patterns over annotations as well as the text. There is a great variety of these search engines available, each displaying specific strengths and weaknesses. We describe three engines as representative examples.

IMS Open Corpus Workbench

IMS Open Corpus Workbench (*CWB*) [77] provides a powerful query language based on regular expressions over sequences of annotations. The engine represents annotated text documents as a token stream to which the majority of features are attached, for example part-of-speech labels or lemma information. Limited support is provided for structural annotations. While it is convenient to search for *all forms of "smoke" and "kill" occurring close to each other* within certain documents or only in headings, recursive structures like constituency parse trees are not well supported. The engine is fast and can be used for large corpora. On a modern computer, a corpus of 100 million words causes absolutely no performance problems. Internally, up to 2.1 billion tokens are supported in version 3.1.

TGrep2

TGrep2 [187] specializes on parse tree structures. It has special query operators related to dominance and child-order. Thus, such queries as *show me all noun-phrases nested arbitrarily deep under a verb phrase* can be easily performed, while they are difficult to impossible to realize with the CWB. On the other hand, regular expression patterns over sequences of annotations, which are easy to realize in the CWB, are difficult to impossible to realize with TGrep2. On a modern computer, searching a parsed corpus of 100 million words is possible, but incurs some waiting.

ANNIS

ANNIS [233] provides sophisticated support for different levels of annotations. For example, part-of-speech tags, lemmata, constituency parse trees, dependency relations, anaphoric relations, etc. can all be stored and queried simultaneously with this engine. However, a corpus of 100 million words is far beyond its capabilities. On a modern PC, around 500.000 tokens should be considered an upper limit for the version 2.2.1.

Definition: ***features*** – A feature is a key/value pair used to annotate primary data. It is usually part of a feature structure, but we also use the term for attributes of annotations in general. A primitive feature has a simple value (e.g. a number or string), whereas a complex feature takes a feature structure as its value. Features are typically typed.

2.2 Automatic analysis

Manual analysis is an expensive process. When huge bodies of text need to be analyzed or when language processing capabilities are to be integrated into devices, automatic analysis is the way to go. Once a sufficiently large body of manually analyzed data is available, it can be used to train statistical models using machine learning algorithms and to evaluate their results against the manually created gold standard. Even for unsupervised machine learning algorithms, i.e. algorithms not requiring any training data, evaluation against a manually created gold standard is indispensable. In a research context, automatic analysis is often embedded into an experimental setup which is iteratively repeated with different variations of analysis components or their configurations to reach optimal results.

When doing manual analysis, annotation guidelines define how to locate the phenomena to be annotated and which categories to use for classifying them. These guidelines are often maintained as simple text documents which can be understood by human annotators, but which are not machine readable. For automatic analysis, the annotation type system plays a similar role. It is a computer-processable, formal document describing the annotation types, their features and tag sets. A typical annotation type system defines a type *Token* which carries a feature *part-of-speech* that assumes a value such as *noun*. Annotation type systems are defined differently, depending on the formalism used to represent the actual annotations, and often depending on the specific analysis tool or processing framework they are written for. Sometimes, standard formats like *XML Schema* [27] or the *Web Ontology Language* (*OWL*) [153] are used to define an annotation type system .

When designing an annotation type system, certain aspects are usually underspecified, in particular such aspects that are explicitly defined in annotation guidelines. Consider the *part-of-speech* feature mentioned above. An annotation guideline should explicitly define which part-of-speech *tags* exist and how they can be distinguished. This set of categories, or *tag set*, constitutes the values which the *part-of-speech* feature may assume. Annotation type systems, however, are often meant to be reusable in different contexts, e.g. for different languages or by scientists of different schools, which categorize parts of speech differently or using a different granularity. Hence, the annotation type system usually defines that the *part-of-speech* feature *exists* on the *Token*, but not the *values* it may assume. If the formalism underlying the annotation type system supports inheritance or some other form of extensibility, some designers may introduce a specialization of the *Token*, e.g. a *FooToken* which accepts only part-of-speech tags from a hypothetical *foo* tag set.

2.2.1 Roles

While manual analysis is mainly done by linguists or other experts from the humanities, persons working on automatic analysis often have a background in computer science or, more recently, computational linguistics. Following Ferrucci and Lally [83], but using more generic names, we distinguish between three roles involved in the implementation of automatic analysis software:

- **Analysis developer** – This role specializes on the implementation of low-level algorithms and their integration with processing frameworks (cf. Section 2.2.2). Theoretic expertise on language is important for this role. The programming skills required for this role are related to the implementation of algorithms that codify this expertise. This role is mostly assumed by computational linguists or computer scientists.

- **Workflow assembler** – This role combines different analysis components to build an analysis workflow specialized for a particular task. This role should mainly be assumed by domain experts, such as linguists.

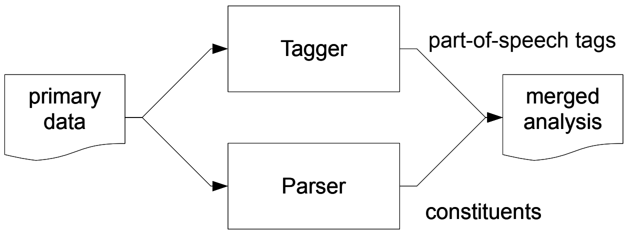


Figure 2.1: Running multiple tools on the same data, merging the output into a single model

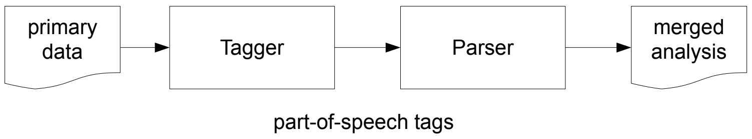


Figure 2.2: Running multiple tools on the same data, one tool building up on the output produced by another tool, merging the output into a single model

- **Workflow deployer** – This role integrates an analysis workflow and the resources required by it into an application or deploys an analysis engine on a compute cluster for high-availability and high-performance processing. This role should mainly be assumed by application programmers or technical staff.

Practically, however, a typical researcher who simply wants to *use* automatic analysis components is often faced with significant hurdles and has to assume each of these roles in turn. While ready-to-use analysis components exist, they can often not easily be combined into an analysis workflow. In the role of an *analysis developer*, the researcher needs to implement additional *glue* transforming data between incompatible analysis components. In the role of the *workflow assembler*, different components then are assembled into a larger analysis workflow. Finally, in the role of a *workflow deployer*, the analysis workflow is installed and executed to gather experimental results. In summary, anybody who wants to employ automatic analysis software requires a considerable set of technical skills – skills that linguists or other researchers from the humanities should not be required to develop.

2.2.2 Processing frameworks

In this section, we introduce the concept of a *processing framework*. This is a piece of software which allows creating interoperable analysis components and facilitates their use.

As the system of language consists of multiple sub-systems, such as syntax or morphology, there are specialized tools and algorithms for these different sub-systems. Often, such tools stand alone – they directly process an input document and produce an analysis result, e.g. TreeTagger [192] can produce part-of-speech, lemma, and chunking analysis. The Stanford Parser [131] can produce part-of-speech tags, syntactic constituents and dependency relations. Both tools come with basic built-in preprocessing capabilities, such as a tokenizer and a sentence boundary detector.

It is desirable to run more than one tool at a time, for example because they analyze different linguistic categories (cf. Figure 2.1). One component can also benefit from the analysis produced by another one. For example, we observed better results in some experiments when, instead of using the Stanford Parser alone, we used TreeTagger to analyze the parts of speech and then forwarded this information to the Stanford Parser to generate the constituency parse tree (cf. Figure 2.2).

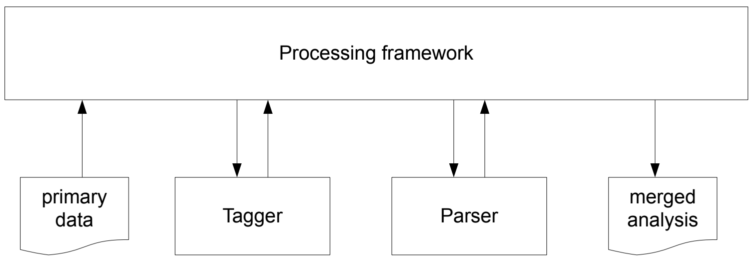


Figure 2.3: Running multiple tools on the same data, integrated within a processing framework, and merging the output into a single model

Most of these stand-alone analysis tools are not immediately interoperable. Each of them consumes and produces different data formats, often even categories at different levels of granularity. Directing the output of one tool into the next one or even just producing an analysis file with the results form different tools is not straightforward at all. So-called *glue code* needs to be written to transform the data exchanged between the tools – time and time again. A processing framework provides a common data model into which analysis results can be written and from which they can be retrieved. Thus, it is not necessary to write glue code for the transformation between arbitrary tool-specific data formats, but only a conversion to and from the common data model (cf. Figure 2.3).

When a tool is integrated into such a framework, we no longer speak of it as a *tool* but rather as an *analysis component*. The integration is realized by implementing a framework-specific wrapper around the tool which takes care of the component's life-cycle (initialization/configuration, execution, destruction), as well as the transformation between the common data model and the tool-specific data format. Once such a wrapper has been implemented, the tool can be used with the framework and be run in combination with other tools in an *analysis workflow*. Some frameworks offer additional features, such as parallel execution or workflow editors, of which the implementer of the wrapper can be oblivious.

A processing framework can provide different levels of interoperability between analysis components. On the lowest level, it provides set of data structures used to store the data to be analyzed and the analysis results, which are used by all analysis components. On a higher level, a framework may offer support for annotation type systems. Consider a *part-of-speech tagger* component that produces an annotation of the type *PartOfSpeech* with a feature *tag* containing the actual part-of-speech tag. A user would expect that a *constituency parser* component should be able to interpret this annotation and use it while constructing a parse tree. Thus, annotation types often become part of the interface specification between analysis components. In component collections, all analysis components should adhere to a common annotation type system. Unfortunately, this also means that components from different component collections are often not interoperable on this level unless the components themselves permit configuring which types they consume or produce.

On yet a higher level, the *constituency parser* not only needs to know about the *PartOfSpeech* type, but also needs to be able to interpret the particular tag set produced by the tagger. Because of the great variety, it is extremely hard for a framework or component collection to provide adequate interoperability at this level. Often, the task of configuring components to be interoperable at this level is left to the user, who would choose, for example, models for the part-of-speech tagger and the parser that were both trained on the same corpus.

The following frameworks have been chosen as general points of reference for the present work, because the component-oriented approach is central to their architecture design. They are not just a collection of algorithms for natural language processing, but explicitly define what an analysis component is, how it interacts with other analysis components and how it can access resources relevant to the analysis. Furthermore, they have all been used to integrate third-party analysis tools from different sources:

- General Architecture for Text Engineering (GATE) [51]
- Text Engineering Software Laboratory (Tesla) [194]
- Unstructured Information Management Architecture (UIMA) [83]

We briefly introduce these frameworks, and their support for the user roles involved in manual and automatic analysis is outlined. All the frameworks target in particular the *analysis developer* role. Each provides an API for building and integrating new analysis components. Hence, support for this role is only discussed if there are special provisions going beyond that.

GATE

The *General Architecture for Text Engineering* (*GATE*) is an instance of an abstract architecture design that Cunningham [51] describes as a *Software Architecture for Language Engineering* (*SALE*). For expert programmers, it provides a well-designed software architecture for processing linguistic data. For non-expert programmers, it provides a graphical environment, the *GATE Developer*, for building research-oriented analysis components and for combining them with each other or with analysis components implemented by expert programmers. While the architecture is general in design, the *GATE Developer* has a focus on information extraction. This is mainly due to *A Nearly-New Information Extraction System* (*ANNIE*) [53], which is the core component that GATE is probably best known for.

The main product, *GATE Developer*, supports in particular the following of the previously introduced roles (Section 2.2.1) in these specific ways:

- **Analysis developer** – The *JAPE* language (*Java Annotation Patterns Engine*) [52], provides a domain-specific language targeted at rule-based analysis tasks, in particular information extraction tasks. In recent versions of GATE, Java can be used in the right-hand side of JAPE rules, making the language much more powerful, but also requiring in-depth knowledge of the interna of GATE.

- **Workflow assembler** – The GATE user interfaces allow conveniently assembling analysis workflows from a set of analysis components that ship with GATE or that are installed from public online repositories of GATE components. Workflows can be exported as *GATE application* files which can be shared with other users. Additionally, workflows can be exported as self-contained packages, including all required primary data and resources, for deployment to the *GATECloud* service [209].

- **Annotator** – A built-in annotation editor allows manually analyzing texts. It is also possible to inspect, and optionally correct, the results of an automatic analysis workflow. As a special feature, the annotation editor even allows making changes to the text. In most language processing frameworks and annotation editors, the primary data is immutable.

- **Explorer** – The *ANNotation In Context* (*ANNIC*) [15] sub-system allows searching a corpus using JAPE patterns. This supports the *corpus explorer* role and also provides a way to quickly test JAPE patterns against real data before they are used in JAPE scripts.

At the time of writing, GATE has evolved to be an established and very successful family of products with an active community, regular tutorials, and commercial users. The suite also includes the full-text search engine *Mímir* [54], the online collaborative annotation and curation platform *GATE Teamware* [30], the *GATECloud* [209] service to deploy analysis services on cloud computing infrastructures, and the minimal framework *GATE Embedded* allows integrating GATE analysis workflows into applications. These additional products also cater towards the needs of the *curator* and *analysis deployer* roles.

Tesla

The *Text Engineering Software Laboratory* (*Tesla*) [194] is a development environment for language analysis based on the *Eclipse platform* [58]. Analysis components and workflows are developed within the Tesla graphical user interface, but they are run on the separate *Tesla Server*. Making intensive use of object-oriented capabilities and providing type-safe APIs was of great importance to the Tesla developers. Another goal was to give non-programmers the ability to visually assemble and run language analysis workflows and to reduce the complexity of implementing analysis components for programmers.

Tesla has put great effort into the implementation of a graphical user interface. It provides two majors modes of operation: the *linguist perspective* and the *developer perspective*. These modes mainly support the following of the previously introduced roles (Section 2.2.1) in these specific ways:

- **Analysis developer** – The developer perspective facilitates the creation of new components and roles. Wizards are provided to generate an initial code skeleton and further code generation facilities are provided to add code for accessing particular data types to a component.

- **Workflow assembler** – In the linguist perspective, the user is presented with an overview over the available components, corpora, and workflows. Components and corpora can be dragged into a visual workflow editor. Inputs and outputs of the components can easily be connected by drawing lines between components. When an experiment has run, its results can be inspected. Tesla exports results to an XML format which can then be processed using XSLT and transformed into input for several visualization modes, e.g. highlighted text, bracketed text, and tabular data. While this is a very flexible approach, the authors of Tesla concede that using XML as an intermediate format has drawbacks, in particular the size of the intermediate XML data can easily exceed hundreds of megabytes even on small corpora annotated for only a few linguistic categories.

- **Workflow deployer** – The *Tesla server* provides a shared repository of analysis components and a means of deploying and running analysis workflows. However, the deployment of new analysis components to the server is not seamlessly integrated into the graphical user interface like many other functions of Tesla.

UIMA

The *Unstructured Information Management Architecture* (*UIMA*) was originally developed at IBM and most prominently described by Ferrucci and Lally [83]. In 2006, the project was donated to the Apache Foundation and remained in incubator status until it was promoted to a top-level Apache project in 2010. The architecture design was also published as an OASIS standard [84]. When mentioning *UIMA* from here on, the Apache UIMA [10] implementation is the intended reference.

The motivation driving the development of UIMA was the integration of language analysis developed at different IBM Research locations into a common framework and to encourage the reuse of components. Thereby, the development of analysis applications was to be sped up and the company to be positioned for a rapid increase of demand for language analysis services.

Support for scaling out analysis across a number of different computers, platforms, or to a cluster is another defining feature of the framework. Platform-independent XML-based descriptors for analysis workflows and analysis components are predominantly used in the framework to facilitate the deployment and interoperability of UIMA workflows. For creating and editing of these descriptors, UIMA provides plug-ins for the Eclipse integrated development environment. UIMA provides several implementations of execution environments which make use of these

XML descriptors, e.g. the *Collection Processing Manager* [81] and Apache UIMA-AS [11]. It also natively supports implementations of analysis components in Java and C++.

The UIMA framework was explicitly designed to support the efficient interaction of people with different skills, such as researchers specializing in linguistic theory and language analysis, domain experts assembling specific analysis workflows for a particular application, and programmers that integrate the analysis into an application or deploy it as a service. UIMA supports mainly the following of the previously introduced roles (Section 2.2.1) in these specific ways:

- **Annotator** – The *UIMA Annotation Editor* is a plug-in for Eclipse which allows creating annotations on text documents according to a configurable annotation type system. The editor can also be used to inspect, and optionally correct, automatically created annotations. An editor for annotation type systems is provided as a separate Eclipse plug-in.
- **Analysis developer** – *UIMA Ruta* (*Rule-based Text Annotation*) [133] targets the *analysis developer* role. It provides a domain-specific language for rule-based language analysis, including an Eclipse plug-in with special editor support and a debugger for the Ruta language.
- **Workflow assembler** – UIMA provides an Eclipse plug-in for assembling analysis workflows. A workflow can itself be used as a reusable component within another workflow, which allows building complex workflows. The form-based editor is powerful, but not easy to use, as it requires quite detailed knowledge of the concepts of the UIMA framework. Assembled workflows and required resources can be exported as a *PEAR* (*Processsing Engine Archive*) for distribution to other users. *uimaFIT*, in turn, focuses on the assembly of analysis workflows in code or scripts, providing a greater flexibility with respect to configuration.
- **Workflow deployer** – *PEAR* files can be installed and run by a workflow deployer in different environments. For example, UIMA provides a simple wrapper exposing a *PEAR* as a REST-based web service.[2] *UIMA-AS* (*Asynchronous Scaleout*) [11] and *UIMA-DUCC* (*Distributed UIMA Cluster Computing*) [217] allow scaling out UIMA workflows to different machines.

uimaFIT / Apache uimaFIT

The strong focus on XML-based descriptors has led to the lack of a convenient API for programmatically and dynamically assembling UIMA workflows, as it is desirable when setting up unit tests or implementing scientific experiments. This gap is filled by *uimaFIT* [171; 186; 14], which provides convenient factory methods which significantly reduce the code necessary to set up a workflow. uimaFIT targets the following of the previously introduced roles (Section 2.2.1) in these specific ways:

- **Analysis developer** – uimaFIT provides a method to maintain analysis component metadata directly in the source code of the analysis components. Parameter values can be comfortably injected into specially marked fields of the analysis component. Convenience

[2] UIMA REST Service: http://uima.apache.org/sandbox.html#simple-server (Last accessed: 2013-05-22)

Definition: *descriptors* – The metadata of an item such as analysis component, analysis workflow, resource, etc. which a processing framework requires to use the item. Sometimes this is also called *description*, in particular in the context of UIMA.

methods allow the type-safe access of analysis results. In summary, analysis components using the uimaFIT facilities can be written with significantly less code than their plain UIMA counterparts.

- **Workflow assembler** – The programmatic assembly of analysis workflows is a cumbersome task and requires verbose code when using the UIMA API. The API provided by uimaFIT wraps the original UIMA API and provides convenience methods that turn the programmatic assembly of workflows into an almost trivial task. It builds on the ability to maintain metadata, such as default parameter values, in the source code of the analysis components. As such, the API greatly facilitates tasks which benefit from assembling analysis workflows at runtime, such as the building of unit tests for analysis components or parameter sweeping experiments which run an experimental setup in many configurations, each of which may require a different workflow.

- **Workflow deployer** – The ability to programmatically assemble analysis workflows also facilitates the integration of such workflows into applications. Likewise, wrappers which allow running an analysis workflow on a compute cluster can be implemented more easily.

uimaFIT has been donated to the Apache Software Foundation in 2012 and was released as Apache uimaFIT 2.0.0 in August 2013. It is now maintained as part of the Apache UIMA project. When referring to *uimaFIT* from here on, it is the Apache uimaFIT [14] which is referred to. In the context of the work leading up to this thesis, various contributions have been made to uimaFIT and Apache uimaFIT including, but not limited to, those detailed in Section 3.2. uimaFIT is also an essential asset used to facilitate the implementation and use of the DKPro Core collection of analysis components for UIMA (Section 5.2).

3 Usability

When talking about *usability*, the first question that needs to be asked is "*What is usable by whom?*" We have previously introduced a scenario in which a linguist and a computer scientist cooperate to analyze a large text corpus with a mixture of manual and automatic analysis procedures. So in this case, the "*what*" are the automatic analysis procedures that the linguist wants to use. The "*who*" are two groups of people, which the computer scientists and the linguist represent in our scenario: *expert programmers* and *non-expert programmers*.

Expert programmers are comfortable working with complex frameworks and employing a wide range of software libraries while writing code in a general-purpose programming language, such as Java.

Experts of a non-computational domain are working on a problem in their domain and require computational assistance. They are not familiar with the details of programming languages and APIs, and they should not have to be. Over the long haul, these domain experts should be able to employ automatic analysis tools without any particular programming expertise. Presently, however, we assume that the domain expert is *just* a non-expert programmer and has some basic programming knowledge, e.g. the ability to write simple scripts. This is not an unrealistic assumption, as students of the, steadily more digital, humanities are increasingly educated in the use of computer-based tools. Harnessing the power of current processing frameworks for the automatic analysis of language, however, is not easily possible using simple scripts.

Given the wide variety of tools and data formats in the NLP domain, combining tools into a processing chain and running it on data from different sources is not trivial. Typically, a solution is either implemented haphazardly for a single purpose or it requires a significant amount of boiler-plate code to build reusable wrappers for embedding analysis tools in a processing framework. Even after these wrappers have been created, depending on the processing framework, additional boilerplate may be required to assemble an analysis workflow. This seriously hinders non-expert programmers in performing such tasks. Efforts need to be made to bring interoperable linguistic processing components to a level at which they can be used by programming beginners or even non-programmers.

A better user interface

When aiming at usability, frequently one of the first steps is adding a graphical user interface to a system. By means of that interface, the non-expert programmer creates some model of a system configuration, which is stored in an intermediate representation, e.g. an XML file. At some point, the user instructs the system to load the system configuration from the file, and to run it. The system then does whatever it has been configured for.

While a graphical user interface is certainly helpful and attractive to the user, the ability to describe the same system configuration in a concise textual description is equally helpful, although less visually attractive. It is like comparing a poem to a picture. In our approach of making automatic analysis useful to the non-expert programmer we prefer to provide a well written poem rather than a colorful picture. A picture may be worth a thousand words, but if one can tell a story in much less than a thousand words, why go through all the effort of painting a picture?

Domain specific languages

Instead of encoding the system configuration in a configuration file, we prefer to employ a *domain specific language* (*DSL*, cf. [19]). Such a language can make the system configuration more concise and human readable than an XML configuration file, and also more flexible. A

DSL targets a very specific task using a focused vocabulary, and is therefore better suited for the non-expert programmer than a general purpose language (GPL). There are two kinds of DSLs:

- **External DSLs** – An external DSL is a new language. It is completely independent from the programming language used to program an application or library integrating the DSL. It comes with its own syntax, requires a special parser, and interpreter.
- **Internal DSLs** – An internal DSL is implemented in terms of another programming language, the *host language*. In effect, a program written in an internal DSL is in every aspect like a program written in the host language, although, it may superficially appear to have a completely different syntax, because the host language is used in unfamiliar ways. Some language properties make a programming language particularly attractive for hosting an internal DSL, e.g. dynamic typing, operator overloading, calling methods using an infix notation (e.g. `1 add 2` instead of `add(1, 2)`, etc.) Certain design patterns are also well-suited for implementing an internal DSL, such as the *builder pattern* [96]. This pattern is used to incrementally construct an object or a specification. The `StringBuilder` of Java is a well known instance of this pattern. Strings are immutable objects in Java. Incrementally building a complex String by concatenating its components can become very inefficient, because every new concatenation operation creates completely new String and allocates new memory. The `StringBuilder` is mutable. It maintains an internal buffer to which data can be added by concatenation operations. Once all data has been added to the builder, it can be transformed into a regular immutable String. This builder supports a *fluent*-style API [89], meaning that most method calls on the builder return the builder itself. This allows writing something like `builder.append("Hello ").append(2).append(" you")`, which is a very distinct style. It can be argued, that an internal DSL is nothing more than an API. This is true. However, it is an API that allows the user to tell a domain story concisely using domain vocabulary and supporting this using distinct design elements, such as the ones just mentioned (cf. [90]).

The benefit of an external DSL is, obviously, that its design can be fully adapted to the domain at hand, while in an internal DSL, the syntax and concepts of the host language will always be present and may pose a limiting factor for the design of the internal DSL. An internal DSL, however, has the advantage of being able to benefit from the full tool chain of the host language, including features of an integrated development environment, such as code completion, refactoring, debugging, etc. at no additional cost.

Declarative programming

Another way to improve the usability of a system is allowing the user to tell the system *what* to do instead of telling it *how* to do it. For example, when querying a database, we use an SQL statement describing *what* kind of data to find by describing the properties of the data we are interested in. The details of how to use database indexes to optimize the response time and how to traverse the index structures to retrieve the data from the database files are left to the database query engine.

We do not want to go the entire way and try setting up a system to which a user only describes what result needs to be produced with the system assembling an adequate analysis workflow to produce this result. In particular, if multiple analysis components are available which could fill the same role, e.g. multiple parsers, we assume that the domain expert, i.e. the linguist, in fact *wants* to choose which one is actually used. We want, however, that these analysis components require little to no additional configuration. Yet, even though configuration should not be mandatory, configuration should be possible.

We consider it an important aspect of declarative programming and of domain-specific languages, that a user can focus on those aspects of a system that need to be customized to reach a specific goal. The user should be oblivious of other details of the system which do not need to be customized. This is particularly important for the domain expert who is neither interested in implementation details, nor, being a non-expert programmer, trained to understand them. It is also convenient for the expert programmer, who can operate on a conveniently high level, but also needs to be able to reach down into the deep implementation details, to implement new functionality, to debug problems, and to optimize the system.

Declarative internal DSL for analysis workflow assembly

We consider a declarative internal DSL the sweet spot of usability for language analysis workflow assembly. A narrowly focused internal DSL allows non-expert programmers to concisely state their domain problems. The expert programmer can easily embed analysis workflows in applications or scientific experiments and can use the full tool chain of the host language.

To improve the declarative aspect of analysis workflow assembly, we reduce the need for explicit configuration. An approach for automatically selecting and acquiring resources required by analysis components removes one of the main needs for explicit configuration and is described in Section 3.1.

The uimaFIT library [186; 171] took a great step towards providing a Java-based internal DSL for setting up UIMA analysis workflows. It provides the necessary vocabulary to conveniently configure analysis components and set up an analysis workflow. Section 3.2 describes how we extend this library to allow the extraction of key behavioral aspects from the component, and thus making them better configurable and extensible. This opens up a completely new way of designing analysis components and analysis workflows, aside from directing output of one analysis component into the next one.

3.1 Resource selection

In this section, we present an approach to the dynamic selection and acquisition of resources required by analysis components during the execution of an analysis workflow.

Our approach simplifies the assembly of analysis workflows, as a component can often be added to a workflow without any further configuration by the user. It provides a just-in-time selection of the resources that allows taking characteristics of the data being processed into account. This is a step towards building analysis workflows declaratively, i.e. by specifying *what* to do, e.g. *run a part-of-speech tagger*, but not *how* to do it, e.g. which model to use, where and how to get it, etc. To enable this, we propose a best practice to package resources as portable artifacts and to distribute them via repositories. This facilitates automatically deploying them to a user's computer, or to another system on which the workflow can be executed, such as a compute cluster. We also suggest an approach packaging resources so that resource artifacts are not specific to a particular processing framework but can be used by different frameworks.

These contributions address the following issues in our overall scenario (Figure 3.1):

❷ **Assembling automatic analysis components into workflows is too complex.**
The automatic selection and acquisition of resources simplifies the assembly of analysis workflows and facilitates this task in particular for non-expert programmers. Thus, it reduces the skills necessary to successfully act in the *workflow assembler* role.

❺ **Workflows are not easily deployable to a compute cluster.**
Packaging resources as artifacts that can be automatically retrieved from artifact repositories reduces the manual effort necessary to set up the execution environment of a workflow on a compute cluster system. It supports the *workflow deployer* role.

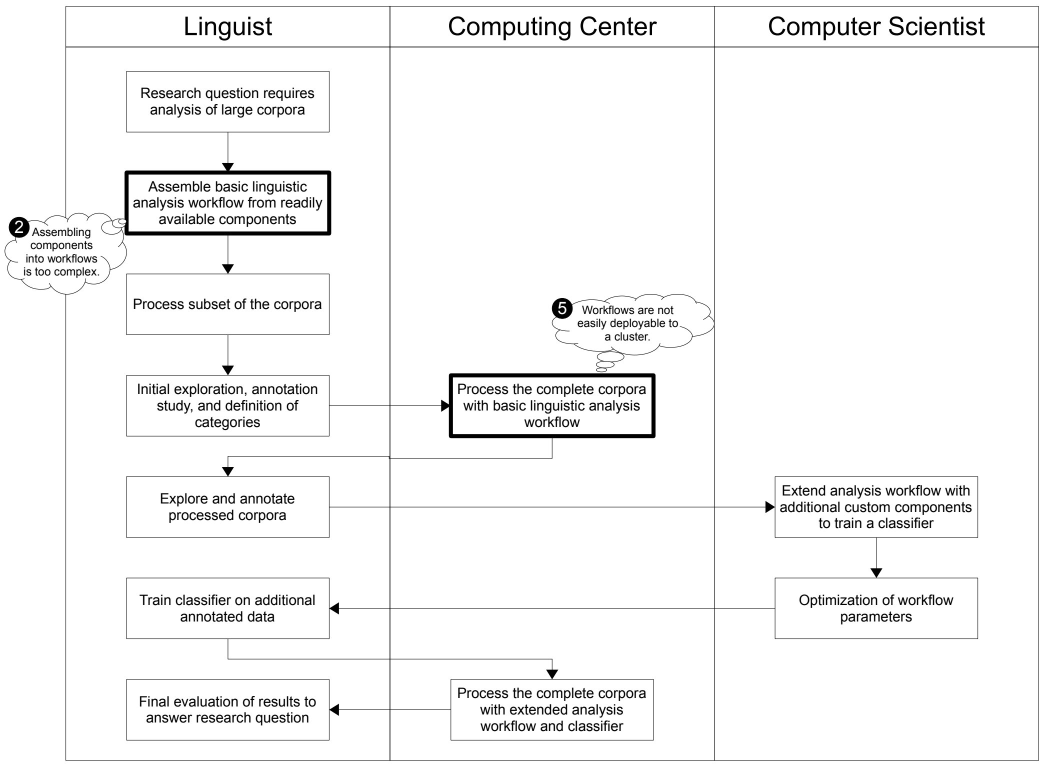

Linguist
Computing Center
Computer Scientist
Research question requires analysis of large corpora
Assemble basic linguistic analysis workflow from readily available components
2 Assembling components into workflows is too complex.
Process subset of the corpora
5 Workflows are not easily deployable to a cluster.
Initial exploration, annotation study, and definition of categories
Process the complete corpora with basic linguistic analysis workflow
Explore and annotate processed corpora
Extend analysis workflow with additional custom components to train a classifier
Train classifier on additional annotated data
Optimization of workflow parameters
Final evaluation of results to answer research question
Process the complete corpora with extended analysis workflow and classifier

Figure 3.1: The automatic selection of resources facilitates the assembly of analysis workflows, whereas the automatic acquisition facilitates deployment, e.g. on a compute cluster.

3.1.1 Motivation

An analysis component often requires a resource, such as a parser or part-of-speech tagger model, a stopword list, etc. A parameter controls which particular resource a given analysis component should use. However, manually locating, acquiring, and configuring every component in a workflow with such a resource is tedious. Being able to produce results quickly with minimal effort is an attractive property of automatic analysis tools. If setting up an analysis workflow and configuring its analysis components is too complex, it becomes unattractive. A framework for natural language processing should allow producing results with minimal or no configuration at all. On the other hand, if configuration is not possible, it also makes a framework unattractive. Likewise, it should be easy to create new analysis components, but the ease should not restrict the power of the created components.

Configuration parameters are used to control the behavior of components and to adjust it to a certain application scenario or experiment. In particular, a parameter can be used to make the component use a specific resource, such as a statistical model, dictionary file, etc. Being able to flexibly configure components is desirable. However, it is tedious and hurts usability when every aspect of a component needs to be configured before the component can be used. There are three approaches how to avoid this:

1. **Sensible defaults** – if a parameter is not explicitly set by the user, a default value is assumed. This default should be the value that most users are expected to have set if they had to. Consequently, it saves the work of explicitly setting this parameter for the majority of the users. For example, it is a best practice nowadays to encode text files in the UTF-8 encoding [231]. The Unicode standard [218] incorporates most written languages, and the UTF-8 encoding is the most popular serialization of this standard. Any component used to read text into an analysis workflow or to write text output should use this encoding by default. It should only be necessary to explicitly configure a component with another encoding in rare cases, for example when an older corpus encoded in a legacy ISO-8859-1 [119] variant is to be processed.
2. **Optional parameters** – an optional parameter does not have to be set. The component is able to cope with this parameter being set to `null` or something equivalent. According to this definition, a parameter with a default value is *not* necessarily optional. It depends on the framework or component implementation if a default value should be assumed when the parameter value is unset. If this is not the case, an error should be generated when a non-optional parameter with a default value is explicitly unset.[1]
3. **Automatic configuration** – the value of a parameter is determined automatically from the information available in the context of an analysis workflow, such as the data being processed, analysis components in the workflow, or the system platform the workflow is running on.

Types of parameters

We can separate the parameters of analysis components into two categories:

- **Data-independent parameters** – these parameters are fixed throughout the execution of an analysis workflow. They are set up when a component is created or (re)initialized.
- **Data-dependent parameters** – these parameters may change during the execution of an analysis workflow. For example, as the result of an auto-configuration mechanism, a parameter may change depending on the language of a document being processed.

[1] While setting up the *DKPro Core* component library (see Section 5.2), we found that practically all parameters are *not* optional, that it is reasonable for a framework to assume the default value of a parameter is unset, and that anything else confuses the users.

Whether a parameter is *data-independent* or *data-dependent* is either the decision of the author of a component or may be prescribed by best practice guidelines for a component collection. For example, one implementation of a parser component may require that the parser model to be used is statically specified (e.g. to a model for a specific language), another implementation may be able to select an appropriate parser model depending on the language of the document being processed (e.g. an English model for English documents, a German model for German documents, etc.). A framework may provide explicit support for data-dependent parameters, by providing a mechanism for specifying different parameter values depending on certain properties of the processed data, for example the language.

Frequently, parameters in analysis components depend on the data being processed, in particular the language of the document. Sensible defaults for such parameters cannot be defined unless parameter values can be defined based on the data being processed. The consequence is that these parameters always need to be defined explicitly, which hurts usability.

Consider a parser component which needs to load a different model file depending on the language of the document being processed. When a German document is being processed, a model for German needs to be loaded, while a model for English is required to process English texts.

Dynamic resource selection and acquisition

In this section, we discuss how to avoid the explicit configuration of analysis components by dynamically selecting suitable versions of those resources that are required for processing and acquiring them. We argue that, with respect to usability, dynamically selecting resources at the run time of an analysis component is preferable to a selection ahead of time. Closely related to the dynamic selection of resources is the problem of actually acquiring the resources and making them available for use by the component. Investigating if and how current processing frameworks support dynamic resource selection and the acquisition of those resources, we find that these concepts are largely unsupported. Therefore, we present our new approach to dynamic resource selection and acquisition.

Taking a look at parameters of analysis components, the kind of mandatory parameter encountered most frequently is the one to select a resource. Such resources may be model files of some statistical tool (e.g. a statistical parser), dictionaries (e.g. stopword lists), or similar resources. Removing the need for explicitly setting these parameters drastically cuts down the manual effort that needs to be put into the creation of an analysis workflow, making the task easier for a non-expert.

Even though, most of the time, there is little actual choice with respect to selecting a particular resource, e.g. because there is only one such resource for the language to be analyzed, the user generally has to explicitly specify this parameter and often even has to manually download the resource and make it available to the analysis component. For the sake of usability, an analysis component should select a default resource suitable to process the data at hand and automatically use it, unless explicitly told otherwise.

Manually acquiring the resource, e.g. downloading it from a website, is a tedious exercise at best. At worst, the user may be unable to locate a suitable resource, because such resources are not always available from the same location as the analysis component or analysis tool they were built for. After selecting which resource to use, the framework should try its best to automatically acquire the resource.

To arrive at a fully configured component depending on one or more resources, several steps need to be performed, as described below. The variation within the process boils down to whether these steps need to be performed manually or automatically and to when they are performed, ahead of time, before any data is seen, or just in time, when data is being processed and its properties can be taken into account:

- **Selecting a resource** – to determine all properties that fully characterize a resource, so that it can be uniquely identified.
- **Resolving the resource** – to determine the full logical or physical address. This can happen by providing this information directly, deriving it from a template by filling in placeholders, or searching in a repository.
- **Acquiring the resource** – to resolve the address and transfer the resource to the local machine so it can be conveniently accessed. Such a transfer may include a download from a remote location. This step is skipped if all access to the resource is done remotely, e.g. if the resource is a web service, or if the resource is already available, for example because it is bundled with the analysis component.
- **Installing the resource** – installing the resource so it can be used by an analysis component. This may include extracting the resource from an archive to the file system, because the analysis component may otherwise not be able to access it. It is possible that different flavors of a resource are bundled, e.g. flavors for different operating systems. In such a case, the resource location may only have to be fully resolved at the time it is installed or accessed.

The following sections describe each of these steps in more detail.

3.1.1.1 Selecting a resource

The choice of resources that can be used with an analysis component is limited by several restrictions.

The first major restriction is made by the analysis component itself. The choice of resources for a particular analysis component is usually quite limited. Most of the time, resources are built for one particular analysis tool or component and are not compatible with any other. Quite often, a resource is even only compatible with one particular version of that component, because there is no backward or forward compatibility. Consider a parser component written in Java. The Java language provides a convenient way of serializing [174] an object graph from memory to a file and at a later point of restoring that graph from the file. When a parser is trained on a corpus, this mechanism makes it easy to persist the learned model to disk. The model becomes a resource which can be distributed to other users so they can use the parser without having to train it. This kind of serialization, however, is very sensitive to changes within the classes from which the persisted objects have been instantiated. When fields have been added, (re)moved, or renamed in a new version of the parser, the old models cannot be used with the new version.

The second major restriction is made by the language of the data being processed. A parser model that has been trained on a German corpus cannot be used to parse English texts.

Of course there are many more restrictions that should be taken into account, in particular regarding conceptual interoperability, but they are often simply ignored. In practice, most of these restrictions tend to be minor, meaning that there is either no further choice, because only one model per tool per language is provided, or because the models are largely technically interchangeable, differing only in coverage, domain, or quality. Take tokenization as an example. A tokenizer component may tokenize the word *don't* in one of four variants: *don't*, *don - 't*, *don - ' - t*, *do - n't*. If the model we choose for a parser has not been trained on the same variant that the tokenizer produces, the analysis results are bound to be erroneous. This, of course, continues with the tag sets consumed and produced by the different analysis components, and applies in general to every aspect in which the components within an analysis workflow need to be aligned with each other. As said before, given that there usually *is* not much choice, these aspects tend to be ignored and errors introduced by such misalignment are taken into account as unavoidable noise. In Section 5.2.4.2, we will discuss conceptual interoperability again in more detail.

The choice for a particular resource may be made fully manually, e.g. making only a single restriction, namely the exact location or identity of the resource, or by otherwise explicitly specifying sufficient restrictions to uniquely identify one resource. The choice, however, can also be made automatically, by taking into account information from the data being processed and from the workflow context. By choosing which component is used in a workflow, the user already made the first major restriction. By processing data in a particular language, the user made the second major restriction. If and how further decisions are necessary needs to be determined on a case-by-case basis. But with these two major restrictions in place, the framework should already be able to select a default resource. It may not produce optimal results for the user, but it should provide *some* results, thus helping the user to get to some baseline results with minimal configuration effort, and in particular without having to explicitly configure the analysis component to use a specific resource.

The selection of a resource can happen *ahead of time*, i.e., before any data is processed. In such a case, an analysis component is restricted to using the pre-selected resource. If a component is able to automatically determine that the resource is not suitable for processing the data at hand, the most it can do is to report an error. If the selection of the resource is deferred until data is actually being processed, the selection happens *just in time*. In this case, information from the data being processed, e.g. the language, or possibly from previous processing steps, e.g. tag set information, can be taken into account.

3.1.1.2 Resolving a resource

Once enough restrictions have been applied so that only a single possible resource remains, this information may be used to derive a location from which the resource can be acquired.

Given the information *an English model for the FooParser specialized on the medical domain*, a user may try looking on the homepage of the *FooParser*, not find anything, and may eventually find one homepage of the *Bar Medical NLP* project. An automatic process may try to use similar information to fill in a location template, e.g. *http://<toolHomePage>/models/<language>/<domain>/model.zip* in an attempt to find the model. As can be seen, it is helpful if there is some fixed scheme by which a resource can be located. It is conceivable, that in future, there may be online repositories of language resources in which such models could be searched for by quite detailed criteria, but at the time this document is written, such repositories do not exist yet.

Resources can be resolved ahead of time if either the resource selection is done ahead of time as well, or if a *set* of resources is defined ahead of time from which the resource selection mechanism may choose. The latter case would typically be combined with actually acquiring the resource, e.g. to bundle resources with an experimental setup or application. Resolving the resource just in time may allow using resources which have not been available at the time the analysis workflow was initially assembled (or started).

3.1.1.3 Acquiring the resource

After the location of the resource has been determined, it is usually necessary to transfer it to the machine on which the analysis workflow will be executed, in order for the analysis component to access the resource. This requires, that the resource is portable and can be transferred. A resource offered via a web service may not be portable. E.g. a web service may offer the ability to query a data source, but not to fully download all data. The next time the service is accessed, it may provide different data or not be available at all. This issue is discussed further in Section 4.1.

Manual acquisition of a resource would typically involve downloading the resource with a browser or copying it from some other location to the local machine. This may be necessary if the resource is protected by some security mechanism which cannot easily be handled automatically, e.g. a single-sign-on system like Kerberos [165] or Shibboleth [160]. Automatic acquisition may also not be possible due to legal reasons, as the user may be required to explicitly accept a license agreement before downloading a resource, which may not be possible in a fully automated scenario, e.g. when automatically downloading resources to a compute cluster. An automatic acquisition process may be as simple as downloading the resource from its resolved location. However, as later discussed in more detail, a resource may depend on further resources which need to be resolved and acquired. Hence, a dependency resolution mechanism may be required for the automatic acquisition.

If acquisition is possible, that is, if the resource is portable, it is typically acquired immediately after it has been resolved. Thus, if the resource is resolved ahead of time, it is also acquired ahead of time, and likewise it is acquired just in time if the resolving is done just in time. Note that acquiring a resource just in time may cause undesired delays in the execution of an analysis workflow. Once a resource has been acquired, it should be cached for further use.

3.1.1.4 Installing the resource

Before eventually accessing the resource, additional steps may be necessary. For technical reasons, an analysis component may not be able to use the resource in the form in which it is distributed. E.g. a resource consisting of multiple files may be distributed as a ZIP archive, but in order for the analysis component to use it, this archive needs to be extracted to the file system. Some resources may not be immediately usable, even as a set of files. For example, a database dump may need to be installed into a database service before it can be used. Binary files may need to be marked as executable. Mind that resources need not necessarily be just data. An analysis component may wrap a tool which can only be invoked as an executable command and not directly via API calls. Thus, the binaries that constitute the wrapped tool may actually be a resource to the analysis component that invokes them.

Parts of the resource selection may have been deferred to this stage. Consider that indeed the binaries of a wrapped analysis tool are about to be installed. The acquired resource is an archive containing binaries for different operating systems and hardware platforms. Depending on what system or hardware the analysis workflow is being executed on, a different binary needs to be installed.

There may again be legal reasons that a resource cannot be installed automatically, e.g. because the installation process requires accepting a license agreement. The automatic installation may also be thwarted by technical issues, such as the need to install a database service before restoring a database dump. In such cases, semi-automatic installation may be necessary, e.g. requiring a manual installation of a suitable database service while automatically restoring the dump. Otherwise, resources can often be installed fully automatically or may even be directly usable in the originally acquired form.

Installing a resource ahead of time may either be necessary because it needs to be done, at least partially, manually. If the resource has been a local one to begin with, e.g. if one particular local file has been initially selected which can be used directly and requires no extraction or further processing, this can also be considered an ahead of time install. A just-in-time installation should be made to a temporary location and should be removed when the analysis workflow is complete. In complex scenarios or when an analysis workflow is embedded within an applica-

Table 3.1: Resource selection (*aot = ahead of time, jit = just in time*)

	Data-dependent selection mechanism	Selection	Resolving	Acquisition	Installation
GATE	static workflow-based `ConditionalSerialController`	aot	aot	aot	aot
Tesla	none	aot	n/a	n/a	aot
UIMA - OpenNLP	static workflow-based `CapabilityLanguageFlowController`	aot	aot	aot/jit	n/a
UIMA - ClearTK	none	aot	aot	component specific	component specific
UIMA - DKPro Core	component-based	aot/jit	aot/jit	aot/jit	aot/jit

tion, the temporary installation may be maintained until the scenario has been fully processed or until the application is shut down.

3.1.2 State of the art

In this section, we examine the support for dynamic resource selection and acquisition in several state-of-the-art processing frameworks, GATE [51], Tesla [194] or UIMA [83]. We find that there is only limited support for workflows to react to characteristics of the data being processed. If support is present, workflows can at most choose between statically configured scenarios. Dynamically looking up applicable resources, acquiring, and using them is not supported by any of the frameworks.

There are several variations on dynamically selecting resources regarding to what part of a system is responsible for the task. This task may either be performed by the analysis component itself or by the workflow controller responsible for the execution of the analysis workflow. The workflow controller determines how data is routed between the analysis components in an analysis workflow, in which order analysis components are invoked, or if they are skipped. The latter case may require adding a component in multiple configurations to the workflow.

UIMA is a special case, as it does not come bundled with analysis components, as the other two frameworks do. Therefore, we also consider analysis components from popular UIMA component collections, ClearTK [172] and the UIMA components from Apache OpenNLP [9]. Table 3.1 provides a comparison of existing frameworks and the approach presented here, which has been implemented in *DKPro Core* (Section 5.2).

3.1.2.1 Component-based selection

In a component-based dynamic resource selection scenario, an analysis component is added once to an analysis workflow. When the component is invoked to process a document, it first analyses the document and then decides which resource is suitable, based on explicitly set configuration parameters and information found in the document. If the data being processed is very heterogeneous, e.g. documents in many languages, the component may have to repeatedly reload resources.

UIMA offers an abstraction layer for resources (cf. Section 3.2) and in particular for *parametrized data resources*. Via this mechanism, an analysis component can request a resource, based on a set of parameters. These parameters are not further defined, but UIMA suggests that the language may be a suitable parameter. However, we did not encounter any UIMA components actually using this mechanism. In future work, we may consider encapsulating our resource resolving mechanism using such parametrized data resources.

3.1.2.2 Parameter-group-based selection

The UIMA framework allows organizing parameter values into named groups. A group name can, for example, be a language code. This provides the possibility to use different parameter values depending on the document being processed. Consider again the parser component. When a German document is being processed, the component can use the model defined in the "*de*" group. For an English document, it uses the one defined in the "*en*" group. Different strategies can be used to resolve a parameter not found in the requested group. The simplest case is to use the value of the parameter declared in a *default* group. Another strategy is aware of language code semantics and can fall back from *en-US* (language/country) to *en* (language) to *default*. Interestingly, there appear to exist no readily available UIMA components making use of this concept.

3.1.2.3 Workflow-based selection with statically configured components

An analysis component may be added to an analysis workflow in multiple configurations, e.g. once configured to use a resource for German and once for English. When the workflow is run, the workflow execution engine examines the document to be processed and then decides which analysis component it should be routed to. Based on the document language, it could decide to route either to the component configured for German or to the one for English. However, this requires that the workflow controller and the analysis components share a common conceptualization of the information relevant to the routing of documents through the workflow, which incurs a potentially undesired coupling between the engine and the components. Such an approach also considerably increases the configuration effort, as configuration variants for all foreseeable cases need to be added to the workflow descriptor. Unforeseen cases can, of course, not be handled at all.

GATE

GATE offers the ability to implement a workflow-based scenario with statically configured components using a conditional workflow.[2] Analysis components within a workflow may be executed or skipped, based on the properties of a document being processed. All resources are selected ahead of time and each analysis component is configured to use a specific resource or combination of resources. Resources are selected based on a fixed name or location. Unless resorting to the scripting mechanisms that GATE offers, the configuration mechanism itself has no concept of selecting a resource based on document properties. GATE tends to bundle resources with its analysis components. Some components (e.g. the Apache OpenNLP [9] components) are also able to load their resources from an URL, which permits a just-in-time installation behavior. In most cases, the analysis components of GATE ship with the resources they need, or, at least, with an initial selection. In a few cases, the user needs to install resources manually, e.g. the platform-specific binaries of external tools.

UIMA

While the UIMA framework has no immediate support for selecting a resource, it offers support for a workflow-based scenario. UIMA assumes that a set of fully specified analysis components are present in a workflow. Each component may declare a set of *capabilities* which indicate that they consume certain types of annotation and certain languages. The framework provides a workflow controller[3] which uses the capability information to route data through

[2] GATE supports a workflow-based scenario using the `ConditionalSerialController` workflow controller.

[3] UIMA supports a workflow-based scenario using the `CapabilityLanguageFlowController` controller.

the workflow. Based on this, workflows can be realized in which certain analysis engines are used or skipped, based on the language of the document being processed and based on the annotation types produced by previously run analysis engines.

OpenNLP

The OpenNLP UIMA components come with partially preconfigured component descriptors. These descriptors already define a language capability, but the actual resources to be used for annotation are not specified. This highlights a problem of the current version of UIMA's capabilities-based workflow. Capabilities are defined statically and cannot change depending on the resources that a component is configured with. Assuming a user configures the OpenNLP's UIMA component for named entity recognition[4] to use a model for Spanish, the user would also need to change the language capability in the component descriptor form "`en`" to "`es`". Resources can only be selected ahead of time by providing their location. The OpenNLP UIMA components rely on the UIMA *data resource* mechanism (cf. Section 3.2), which supports loading a resource from an URL. The resources are loaded automatically by UIMA at the time the analysis workflow is initialized. They cannot react to changes in the data, e.g. to documents in different languages. Since OpenNLP is able to load all resources from streams, no installation is required.

3.1.2.4 Workflow-based selection with dynamically configured components

Instead of initializing and instantiating all analysis components when an analysis workflow is started, the workflow controller could defer this until the workflow is actually started and the data being processed is known. In this case, the analysis component descriptors are templates with placeholders, which are filled in by the workflow controller using information from the document being processed, before instantiating the analysis component to process the data. Again, this would require introducing a potentially undesired coupling between data, analysis components and workflow. It may also be problematic to model cases where a component loads multiple interdependent resources. In a component-based scenario, the component-specific information which resources interact with each other and how, can easily be modeled within the component itself. In a workflow-based scenario, an extra mechanism would need to be introduced to communicate such interdependencies to the workflow controller.

We did not encounter this approach in any of the examined processing frameworks. However, in Section 4.2, we contribute an approach for dynamically generating analysis workflows.

3.1.2.5 No dynamic selection

Tesla

Analysis workflows built with Tesla are meant to be able to run on a dedicated Tesla Server. Since no resource acquisition mechanism is integrated in Tesla, this limits the choice to those resources bundled with the analysis components deployed on the Tesla Server. Dynamic resource selection is not implemented in Tesla, as neither the components support a corresponding mechanism, nor is a conditional workflow available.

ClearTK

ClearTK does not provide descriptors for its UIMA components. Instead, it relies heavily on uimaFIT to programmatically generate descriptors, set parameters, and configure resources. ClearTK components are configured ahead of time with resource locations. Although some components also require or allow specifying the language as a parameter, this information is

[4] OpenNLP's UIMA component for named entity recognition is implemented in the `NameFinder` class.

not used to resolve the models. A workflow-based mechanism, e.g. via the capability-based workflow controller, is not supported as the ClearTK components do not declare capabilities. Since the capabilities can easily change depending on the resources a component is configured with, a mechanism would be required by which uimaFIT could determine a component's capabilities based on its parametrization while generating the component descriptor. Such a mechanism, however, is not available. Most resources are bundled with ClearTK components. It depends on the individual components if they allow downloading a resource file from an URL or if only locally available files are allowed. There is no particular sub-system in ClearTK which takes care of acquiring remote resources, and UIMA's data resource mechanism is also not used. Likewise, there is no particular mechanism to install resources for ClearTK components. Some components require that their resources are on the file system, others can directly access the bundled resources. The way this is handled differs from component to component.

3.1.3 Contribution: Dynamic resource selection and acquisition

In an effort to remove the necessity of explicitly specifying which resources an analysis component should use, we developed an approach for selecting, resolving, acquiring, and installing resources automatically and dynamically depending on the data being processed. Analysis components employing this mechanism provide a better usability for non-expert users, because the main need for configuration has been removed and the components "*just work*." This section introduces the approach and various conventions for addressing as well as for packaging the resources and making them portable:

- A coordinate system for resource selection
- A resource selection and acquisition process
- A best practice for packaging resources

3.1.3.1 A coordinate system for resource selection

As mentioned before, typically a few pieces of information are critical to selecting the proper resource. We shall call these the coordinate of a resource:

- **Type** – if the resource is specific to a tool, the *type* specifies the tool, e.g for the Stanford CoreNLP [200] suite, there are resources for the *parser*, *tagger*, etc. Otherwise, if the resource is generic and may be used by multiple tools, this coordinate indicates the kind of resource (e.g. *gazetteer*, *stopword list*, *ontology*, etc.).

- **Language** – the *language* is important, as most resources are language-specific. In case a resource is not language-specific, some well known value can be used here. ISO-639-2 [121] defines the language code *und* for *undefined*. For resource artifacts containing binaries, this may just be left empty.

Definition: *coordinate* – A scheme for addressing data based on key-value pairs. Maven employs GAV (*group*, *artifact*, *version*) coordinates to address artifacts. In this work, we propose a base coordinate system consisting of *type*, *language*, *variant*, and *version* for addressing resources needed by analysis components, e.g. models, dictionaries, etc. Additional coordinates like *platform* or *tag set* may be used to further qualify certain kinds of resources, such as platform-specific binaries or annotation type mappings.

Table 3.2: Exemplary assignment of resource coordinates

Resource	Coordinates			
	Type	Language	Variant	Version
OpenNLP part-of-speech tagger model	opennlp-model-postagger	en	maxent	1.5
OpenNLP part-of-speech tagger model	opennlp-model-postagger	en	perceptron	1.5
CoreNLP parser model	corenlp-model-parser	en	factored	1.3.5
CoreNLP parser model	corenlp-model-parser	en	pcfg	1.3.5
CoreNLP named entity recognizer model	corenlp-model-ner	en	all.3class.distsim.crf	1.3.5
CoreNLP named entity recognizer model	corenlp-model-ner	en	conll.distsim.crf	1.3.5
CoreNLP named entity recognizer model	corenlp-model-ner	en	muc.distsim.crf	1.3.5
TreeTagger part-of-speech tagger model	treetagger-model-postagger	en	default	3.2
TreeTagger chunker model	treetagger-model-chunker	en	default	3.2
TreeTagger binaries	treetagger-bin	–	–	3.2
...	...	...	...	...

- **Variant** – there may be arbitrary additional information necessary to disambiguate a resource if multiple resources for the same tool and language are available, which we summarize under the *variant* coordinate. For models of probabilistic analysis components, the variant can encode the most important parameter values used to train a model. For resources containing only language-independent binaries, this may be left empty if binaries for all platforms are bundled together. Otherwise, this may be used to indicate the platform (e.g. *linux-x86_32*, *linux-x86_64*, etc.).

- **Version** – each resource should have a *version*. A resource may be improved over time or needs to change in order to function for newer versions of the associated tool.

Although, it is not quite correct, we assume that the language implicitly specifies additional characteristics, such as the tag set (cf. Section 5.2.4.2), tokenization scheme, etc. In order for a user to get results quickly and conveniently these coordinates suffice. In the present scheme, any distinctions that need to be made beyond the type and the language, should be encoded in the variant.

In practice, this set of coordinates has proven to provide considerable convenience to the users of the DKPro Core component collection (Section 5.2). Table 3.2 illustrates resource coordinate as they are being used in DKPro Core.

Default variant

If there exists more than one resource per type and language, then the *variant* is an additional coordinate used to disambiguate a resource. As not to force the user to choose explicitly between variants, every component should consider one variant the default, which is used unless anything else is explicitly specified.

Which variant of a resource is suitable can be completely independent of the data being processed. Instead, variants of a resource typically provide a choice between quality, memory requirements, and processing speed. For example, the Stanford CoreNLP parser can operate with two kinds of models: a *factored* model [131] or a *PCFG* model [132]. While both models are trained on the same data, the factored parser requires more memory and is slower than the PCFG parser, but tends to produce better results. Thus, when quality is required, the *factored* variant of the model is used, otherwise the *pcfg* variant (cf. Table 3.2).

The default variant should provide a good balance between quality, memory requirements, and processing speed. It should also be based on the most commonly used data set or tag set for the language. For example, for the English language, default variants should prefer the Penn Treebank [146] part-of-speech tag set, syntactic categories from the Penn Treebank, and so on.

There may be a different variant for every language suitable for being taken as the default variant. In order to avoid explicitly labeling one resource as the *default* variant, we employ several techniques:

- **Fixed default** – in cases where variants are consistently available across languages, the analysis component uses a hard-coded default variant. For example, the OpenNLP part-of-speech tagger models consistently come in a *maxent* and a *perceptron* variant, so we can choose one of them to be the default.
- **Language-dependent default** – in cases where for each language a different variant is more suitable, we use a mapping assigning the default variant based on the language. For known resources, this mapping is bundled with the analysis component. If a user has custom resources or has a different preference for the default resources, this mapping may be overwritten.
- **Default redirect** – both previous approaches assume that the default variants are known when the analysis component is created. To permit a component in a just-in-time resolving scenario to automatically support resources for new languages as they become available, a *proxy* resource with a well known variant (e.g. *default*) can be introduced. Then, the component resolves this default proxy resource, the resolving mechanism is redirected to the actual resource.

3.1.3.2 Resource selection and acquisition process

Our process for selecting and acquiring a resource consists of four steps which are described in this section:

- **Resolving the resource artifact** – Determining the artifact containing the resource based on the artifact coordinates.
- **Resource artifact acquisition** – Fetching the artifact.
- **Resolving the resource within the artifact** – Determining the location of the resource within the artifact.
- **Resolving auxiliary resources** – Determining any auxiliary resources which are required, e.g. metadata describing the resource.

Resolving the resource artifact

The resource coordinates can be used to construct the information necessary to access a resource. An example how an URL template making use of these coordinates may look like is given in Figure 3.2.

```
http://repository.org/${type}/${language}/${version}/${variant}/resource.jar
```

Figure 3.2: URL template using resource coordinates

However, we do not rely on the resources to be available at a particular URL. Since there are currently no repositories on which we can operate directly with resource coordinates, we map the coordinates to create a new set of coordinates, so called GAV (group, artifact, version) coordinates, as they are used by Apache Maven (or just Maven for short [156]).

Maven employs a declarative approach for building software. In particular, it allows declaring dependencies on external libraries and allows automatically resolving such dependencies by downloading them from a repository and making them available to the locally running build.

Table 3.3: Example mapping resource coordinates to Maven GAV coordinates

Coordinate	Value	Mapping
groupId	`de.tudarmstadt.ukp.dkpro.core`	
artifactId	`corenlp-model-parser-en-pcfg`	`${type}-${language}-${variant}`
version	`1.3.5`	`${version}`

The mechanism is not limited to software libraries, but can be used to resolve any kind of *artifact*. Artifacts are identified by a *groupId*, broadly specifying the organization or project context, the *artifactId* naming a particular artifact within the context, and a *version*. For example, the UIMA core library has the GAV coordinates *org.apache.uima : uimaj-core : 2.4.1*. Maven is supported by a global infrastructure service, the Maven Central Repository [149], to which artifacts, mainly Java libraries, can be uploaded. There are many public third-party repositories which a user may opt to use. Finally, it is easy to host an own private or public repository using one of the different Maven repository managers, like Apache Archiva [7] or commercial alternatives. When a build is performed, Maven resolves the GAV coordinates against all registered repositories and automatically downloads all required artifacts and their transitive dependencies.

For the analysis tools integrated into the DKPro Core component collection (Section 5.2), we have packaged a considerable number of resources for different tools and languages as Maven artifacts and distribute them via a public Maven repository. DKPro Core also includes scripts which automatically download the resources from their original providers and package them according to the conventions outlined here and implemented in the DKPro Core component collection. This way, a user can easily package all resources, in case this repository is not available. The scripts can easily be updated when newer versions of the resources are released.

Resource artifact acquisition

If the artifact containing the desired resource should be automatically acquired, we can benefit from the Maven infrastructure, by translating the resource coordinates into GAV coordinates (see Table 3.3).

With this information, the resource artifact can be automatically acquired. This can either be done ahead of time, e.g. by specifying the resource as a dependency in the Maven build information for the project which embeds our analysis workflow, or the resource could be acquired just in time, when a component actually needs the resource. The latter requires that the Maven dependency resolution mechanism (or an alternative) be integrated into the component.

Resolving the resource within the artifact

After the resource artifact has been acquired, it is necessary to locate the actual resource within the artifact. The acquired artifact is not the resource per-se. It is an archive that may contain one or more resources and the corresponding metadata.

A second location is derived from the resource coordinates, this time to locate the resource within the artifact. Note that we assume here that every artifact contains only a single specific version of a resource. Hence, the version information is not used here (Table 3.3).

```
/de/tudarmstadt/ukp/dkpro/core/lib/${type}-${language}-${variant}.bin
/de/tudarmstadt/ukp/dkpro/core/lib/${type}-${language}-${variant}.properties
```

Figure 3.3: Template to locate a resource within a resource artifact

While the information like the language information is typically determined from the data being processed, there is other, runtime-specific information which may be necessary to resolve a resource. For example, knowledge about the system platform on which an analysis workflow is running may be necessary to choose the correct binaries for that platform (e.g. *linux-x86_32*, *linux-x86_64*, etc.).

```
/de/tudarmstadt/ukp/dkpro/core/bin/${type}/${platform}/
```

Figure 3.4: Template to locate binaries within a resource artifact using the system platform as an additional coordinate

Resolving auxiliary resources

This time, additional, component-specific information may be taken into account. Consider a part-of-speech tagger component which selects a tagging model for English. The component employs a mapping to determine as which annotation type a particular part-of-speech tag should be rendered. E.g. the annotation type system may include a *PUNC* annotation for punctuation, while the selected part-of-speech tagger model produces a "." tag. The model can declare which tag set it uses as metadata in a *.properties* file associated with the resource, e.g. as `pos.tagset=ptb`.[5] Thus, after selecting the model, the component can use the tag set information as an additional hint to select the annotation type mapping.

```
/de/tudarmstadt/ukp/dkpro/core/lib/${language}-${pos.tagset}-pos.map
```

Figure 3.5: Template to locate an auxiliary resource (an annotation type mapping) using additional coordinate obtained from a primary resource (a part-of-speech tagger model)

Resource installation

In the best case, a resource can be accessed directly from the acquired artifact, without any further steps. To unify the access to the resource within the artifact, we employ the Java *classpath* mechanism. Similar to URLs, the classpath provides a uniform address space. To make software libraries accessible by a Java application, the JAR files or file system folders containing the compiled libraries are mapped into this address space. JAR files are basically ZIP archives and may contain any kind of data including resources that may be used by analysis components. Consider Maven is being used to build a software project embedding an analysis workflow. Any resource artifacts added as dependencies to this project are automatically added to the classpath. Since any data in the classpath can be accessed via an URL or as a stream, any component supporting these modes of access can immediately use the resources stored in these artifacts using classpath locations such as previously illustrated in Figures 3.3-3.5.

Should an analysis component not be able to load resources from the classpath, the resources need to be extracted to a temporary location in the file system. This may be necessary if a component wraps an external command-line analysis tool. If the resource is a single file within the resource artifact, it can easily be copied to a temporary file. If the resource consists of multiple files, a more sophisticated approach needs to be taken. As a best practice, all files belonging to the resource should at least be within the same folder inside the resource artifact. Mechanisms exist to recursively examine a folder in the classpath and extract it to the file system.

Care should be taken that resources, just like class files, reside in unique folders within the classpath. Thus, a fully qualified name starting with a name *owned* by the resource provider, e.g. `/de/tudarmstadt/ukp/dkpro/core/resource/...`, should be used to avoid conflicts. Further, no two resource artifacts should use the same prefix. Consider the two resource artifacts `english-resources` and `german-resources` containing a parser model at

[5] *ptb* is here short for *Penn Treebank*

`/resources/parser.ser`. When these two artifacts are mounted to the classpath, both, the German `parser.ser` and the English `parser.ser` are mapped to the same address. It is absolutely not trivial for an analysis component to deal with such a case and choose the correct resource to use. Unfortunately, some of the few resources already distributed via Maven repositories do not observe these best practices yet, which makes it problematic to use them, even though it should be possible to integrate them easily into any framework via proxy artifacts.

Discussion of resource resolving and acquisition

While implementing this dynamic resource resolving and acquisition strategy, it became obvious that determining which version of a resource is compatible with a specific version of an analysis component is not trivial, in particular for a non-expert user. Tools and their resources may have different release cycles. Additionally, analysis components wrapping particular tools often have different release cycles than the tools they wrap.

To address this issue, we record the versions of all known compatible models in a metadata file[6] shipped with the analysis component. When a component tries to access a resource, it can refer to this information to determine a compatible version. If no version is found, a component could try to acquire the *latest* available version of the resource. However, since the *latest* version is bound to change over time, the reproducibility of results is not guaranteed. Also, the *latest* version may not be compatible with the present version of the analysis component.

As mentioned before, our approach relies on the Maven infrastructure. If Maven is actually used as a build tool, resources may be added directly as dependencies to the project embedding the analysis workflow. That way, Maven resolves and acquires all resources ahead of time. However, this requires that the set of resources that will be required during the execution of the workflow is known. If resources for German and English are added as Maven dependencies, these are automatically installed ahead of time. If a component then processes Spanish documents, it has either to try acquiring the resource just in time, or it has to produce an error message. A scientist who desires full control over the experimental setup and the resources used may want to prevent the component from doing automatic downloads at runtime and restrict the resources to those specified as Maven dependencies.

3.1.3.3 Packaging resources for reuse

A resource may be completely stand-alone, possibly encoded in a widely-used standard format, so that it can be used in many contexts, for example a dictionary, ontology, or other lexical resource. More often, however, a resource originates from the context of a particular tool and is specific to that particular tool implementation. For example, a model trained for a particular implementation of a parser may be saved as a serialized representation of the classes used in this implementation. Since this is a convenient strategy, it is used often. Such a serialized model can typically not be used by a different parser implementation and sometimes not even by a different version of the same parser.

A framework can use a resource more effectively, if it can make use of metadata about the resources. A resource in its original context often bears little to no metadata about itself. For example, it may not be possible to determine if a parser model has been trained on a German or an English corpus, unless that information is explicitly stored in a metadata file which the framework can access. Just as there are no readily usable resource repositories, there is likewise no agreed-upon standard to represent such metadata. Hence, processing frameworks or component collections tend to define their own specific metadata formats, in which they are

[6] In fact, we use the Maven *project object model* (POM) file to store this information in the form of so-called *dependency management* information. This instructs Maven which version of an artifact to use without actually declaring a dependency.

also free to add any non-standard information. We do acknowledge the existence of many metadata schemes. Unlike efforts such as CMDI, we do not attempt to integrate these with each other. However, we rather suggest an approach by which each framework may maintain resource metadata, but the bulk of resources may still be used by analysis components of different frameworks.

Resource artifacts may be of significant size. Repeatedly packaging resources along with framework specific metadata in framework specific artifacts and distributing these via a repository infrastructure such as Maven Central is extremely inefficient. Resources, as software libraries, should be packaged and provided in a way that makes them easily usable by any framework or even by stand-alone tools. Framework-specific resource metadata should be maintained separately from the resources. It should also be maintained separately from the components that use the resources, because resources, tools, and the analysis components that wrap them, may evolve at different speeds. So it is possible that one version of a resource may be usable by several versions of an analysis component, e.g. if the version of the wrapped tool remains the same.

Four tiers of artifacts

Currently, resources are seldom packaged and distributed separately from tools or analysis components. Instead, they are included in the package of the processing framework or of an analysis component. Even when they are packaged as separate distributable artifacts, there may be other good reasons not to use these packages, such as the package granularity. Including all upstream resources distributed with a tool into a single resource artifact may result in a *huge* package. Consider a statistical parser for which models for various different languages exist, each model easily reaches tens of megabytes in size, some reach even a gigabyte or more. A user interested in processing one language only should not be forced to download the models for other languages.

For the optimal reuse of resources, we propose a four-tiered approach to packaging resources:

- **Upstream resource** – unpackaged resource as it was originally provided by the creator. The upstream resource may not be addressable or versioned in any way.
- **Resource artifact** – resource packaged in a way that it can be distributed via a repository. In addition to any metadata required by the packaging and distribution mechanism itself, the resource artifact should contain information about its creation, about the origin of the resource, and about its license.
- **Proxy artifact** – framework-specific metadata for the resource. The proxy artifact refers to the resource artifact.
- **Component artifact** – analysis component which can make use of the resource. The component refers to the proxy artifact.

To realize this approach, we assume that an artifact bears a unique identifier and a version so it can be addressed, e.g. by means of GAV coordinates. We further assume that it is possible to declare that one artifact depends on another, such as the Maven dependency declaration and resolving mechanism.

Figure 3.6 illustrates the relationship between a resource, its original tool context and analysis components from different component collections. Consider a resource from the context of the *Baz* tool. To make the resource available to analysis components based on the two processing frameworks *Foo* and *Bar*, it is first packaged in a *resource artifact*. Any resource metadata a framework may require is packaged in a separate artifact, a *proxy artifact*. The proxy artifact declares a dependency on the resource artifact. Finally, an analysis component which can

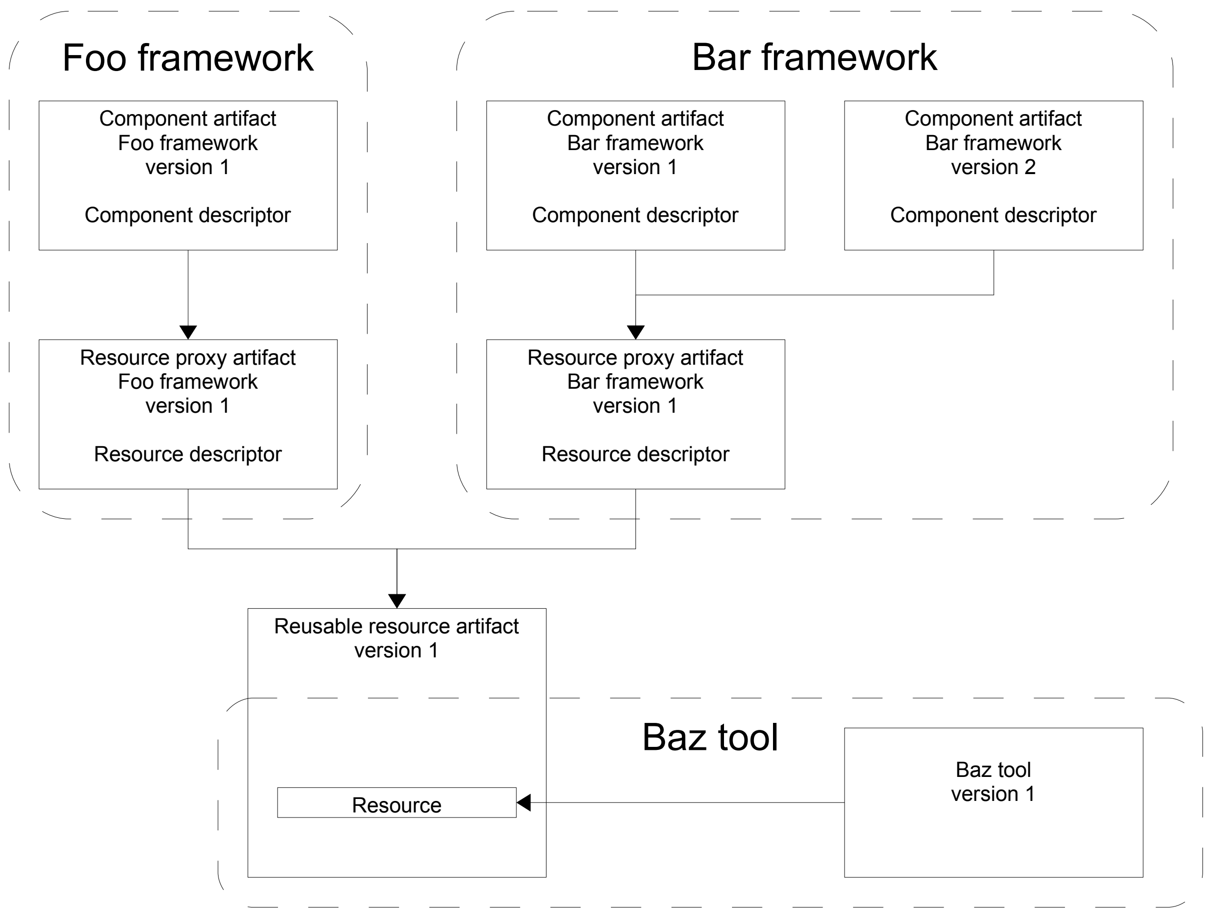


Figure 3.6: A resource and its relations to the original tool and to analysis components from different processing frameworks

use the resource is packaged in a *component artifact*. The component artifact declares a dependency on the proxy artifact. To truly decouple the analysis component from the resource, it is necessary, that the analysis component does not try to access the resource directly within the resource artifact. Instead, it should use a pointer stored in the proxy artifact to resolve the actual resource.

This four-tiered approach has several benefits:

- **Reusability across frameworks** – The resource artifact can be used by different processing frameworks alike, assuming that each framework supports the packaging format and possibly the logical identifiers and the distribution mechanism.
- **Reusability within framework** – The resource proxy artifact can be used by different components within the same framework. As previously indicated, the proxy can also be used to implement a redirection from a *default* variant proxy to the actual default resource. Since multiple proxies can be set up for the same resource, the resource can have the *default* variant coordinate in addition to its actual variant coordinate.
- **Reusability across versions** – The processing framework and its analysis components can evolve independently from the resource. When the metadata required by the framework changes or is extended, it is only necessary to create a new version of the resource proxy artifact which contains the metadata. This is beneficial, because the resource proxy artifact is typically very small while a resource artifact tends to be very large.

Decoupling resources from components

An analysis component should not declare a dependency on any resource. Such an explicit dependency would entail that the resource is always acquired and potentially bundled with an application, even though it may never be used, leading to unnecessarily large application bundles. It should be left to the user or the automatic resource selection mechanism to determine which resources an analysis component should use. A user who wants to select the resources to be used by an analysis workflow ahead of time, can add dependencies on the resource proxy artifacts to the workflow. It is helpful, though, if an analysis component declares with which resources it is compatible. Maven, for example, offers a *dependency management* mechanism, which allows declaring which version of a particular artifact should be used *if* that artifact should be added as a dependency. In that way, a component may declare its compatibility with certain proxy artifacts, with the metadata contained therein and, thus, indirectly with the resources these proxies depend on.

It is even possible to operate altogether without any explicit references to the proxy. Assuming the proxy artifacts use identifiers following a certain scheme, the analysis component could construct a proxy identifier at the time that data is actually processed, based on the data that is being processed. The component can then resolve this identifier and download the proxy and resource artifacts from a repository at runtime. If a *soft* reference to compatible resources is available in the component artifact, this can be used to choose a resource version, otherwise the component may try to fall back to looking for a proxy artifact with the same version it has itself, or try looking for the latest version of the resource.

Resources required for analysis components such as parser models or dictionaries can be loaded from the Java class path. This allows packaging such models as JARs, publishing them to artifact repositories and using them via Maven. Maven's dependency resolution mechanism can then be used to automatically download an artifact and all its dependencies.

It is not necessary to declare the required resources at build time. The framework incorporates support for resolving resources at run time.

3.1.4 Example

As a proof of concept, we have implemented the dynamic resource selection and acquisition approach described here for the DKPro Core component collection (Section 5.2). Components that make use of this mechanism define a specialization of the abstract `ResourceObjectProviderBase` class. For an illustration, we examine the implementation of the DKPro Core wrapper for the part-of-speech tagger component from the Stanford CoreNLP library. Listing 3.1 shows how the component configures the resource provider. First, a configuration file for the language-dependent default *variant* coordinate is provided. Then, the location of the resource proxy metadata for the part-of-speech model is configured. Finally, the GAV coordinate template for the resource proxy is provided.

Listing 3.1: Default information for the resource provider used in the DKPro Core StanfordPosTagger component

```
setDefaultVariantsLocation(
  "de/tudarmstadt/ukp/dkpro/core/stanfordnlp/lib/tagger-default-variants.map");

setDefault(LOCATION, "classpath:/de/tudarmstadt/ukp/dkpro/core/stanfordnlp/lib/" +
  "tagger-${language}-${variant}.properties");

setDefault(GROUP_ID, "de.tudarmstadt.ukp.dkpro.core");
setDefault(ARTIFACT_ID,
  "de.tudarmstadt.ukp.dkpro.core.stanfordnlp-model-tagger-${language}-${variant}");
```

The component offers several parameters to allow the user to override the language, variant, or even the location of the model file, allowing to selectively or fully disable the automatic selection mechanism. Listing 3.2 illustrates how these override parameters are passed on to the resource provider.

Listing 3.2: Resource selection overrides in the DKPro Core StanfordPosTagger component

```
setOverride(LOCATION, modelLocation);
setOverride(LANGUAGE, language);
setOverride(VARIANT, variant);
```

The resource proxy metadata contains framework-specific information as well as a pointer to the actual resource location. Listing 3.3 illustrates what this metadata can look like.

Listing 3.3: Resource selection overrides in the DKPro Core StanfordPosTagger component

```
generated=2013/05/08 00\:24
version=20130404.1
language=de
variant=fast
location=classpath\:/de/tudarmstadt/ukp/dkpro/core/stanfordnlp/lib/tagger-de-fast.tagger
pos.tagset=stts
```

The *ResourceObjectProviderBase* implements all functionality to monitor changes to any parameters that may affect the resolved location of the resource. Whenever any of these parameter changes, it resolves the resource location and passes it as an URL to the abstract `produceResource` method. This method is implemented specifically for each component to reload the resource. Listing 3.4 shows the implementation for the Stanford part-of-speech tagger. Note, that a new part-of-speech tagger instance is created only when the resource URL changes as a result of the new parameter values.

Listing 3.4: Resource access in the DKPro Core StanfordPosTagger component

```
MaxentTagger produceResource(URL aUrl) throws IOException {
  return new MaxentTagger(aUrl.toString());
}
```

Whenever the component processes a new document, the resource provider is updated with data-dependent parameters, such as the language. The component does not hold a reference to the part-of-speech tagger. Whenever it requires one, it calls the `getResource` method. This may trigger a reloading of the resource and the creation of a new part-of-speech tagger, if a data-dependent parameter caused the resolved resource location to change. Listing 3.5 shows how the part-of-speech tagger is acquired from the resource provider and used to tag the words in a sentence.

Listing 3.5: Resource access in the DKPro Core StanfordPosTagger component

```
words = modelProvider.getResource().tagSentence(words);
```

The user feedback we got after integrating our approach into the DKPro Core component collection (Section 5.2) was very positive, in particular because the system always chooses a version of a resource compatible with the analysis component being used. Users manually choosing incompatible versions of resources had been a frequent problem before.

3.1.5 Summary

In this section, we have motivated the need for the dynamic selection and acquisition of resources and found that this concept is not supported by current processing frameworks. With present frameworks, resources need to be selected and mostly acquired ahead of time, before the analysis workflow is executed.

To address this, we have introduced a new approach to dynamically select and acquire resources. Our approach implements a just-in-time selection, which allows taking characteristics of the data being processed into account which are only available at runtime. With this approach, we removed the need for explicitly configuring analysis components the most frequent mandatory parameters, the required resources. In most cases, components can be used in an analysis workflow without any further configuration, which significantly simplifies the assembly of the workflow, in particular for non-expert programmers. When desired, the default configuration may be overridden, e.g. to choose a different variant of a resource or to use a custom local model instead of a model from a repository.

Our approach represents a step towards building analysis workflows declaratively, by specifying *what* to do, e.g. *run the OpenNLP part-of-speech tagger*, but not *how* to do it, e.g. which model to use, where and how to get it, etc. We expect this to facilitate the development of declarative domain specific languages for the assembly of workflows, as well as interactive tools for workflow assembly, and annotation tools which embed automatic analysis workflows.

An integral part of the approach is the consistent packaging of resources as resource artifacts which can be distributed via artifact repositories. Packaged resources are portable, which allows easily and automatically deploying them to a user's computer, or to another system on which the analysis workflow is executed, such as a compute cluster.

We have suggested a best practice for packaging resources in a way that is not specific to a certain tool or processing framework. Instead, the resources can be reused, avoiding doubts if two independent packagings of a resource are actually the same, and saving space in the repositories.

As we notice that several parties have interest in or are starting with the distribution of resources via Maven repositories, we hope that our guidelines will help to avoid redundant packaging of resources, as we can already witness it. E.g. some models of the Stanford CoreNLP [200] suite are already packaged and distributed multiple times and in different artifacts and using different packaging strategies via Maven Central.[7]

Future work should provide a more refined set of coordinates and possibly an enhanced repository which allows searching by these coordinates. For example, we currently assume that the language implies additional characteristics, such as the tag set, tokenization scheme, etc. While this is a viable assumption, as we discuss in Section 5.2, it is not valid in general. However, for a user to get results quickly and conveniently, these coordinates suffice. In the present scheme, any distinctions that need to be made beyond the type and the language, should be encoded in the variant. Promising work that might lead to such a refined coordinate set and associated repositories includes, for example, *ISOcat* [130], OLiA [43], the *Component Metadata Infrastructure* (*CMDI*) [34] and the CLARIN Virtual Language Observatory [223].

Based on the positive experience we collected with our approach, we are planning to extend it to support the dynamic acquisition of primary data, e.g. text corpora. E.g. in a research group, corpora could be stored on a server and be conveniently addressed and used by the researchers in their experiments. The NLTK framework [26] provides a convenient mechanism for dynamically acquiring primary data, which might serve as a point of reference.

[7] See e.g. *edu.washington.cs.knowitall.stanford-corenlp* model artifacts and the *edu.stanford.nlp* model artifacts, which can be found by searching for "*stanford models*" on `http://search.maven.org` (Last accessed: 2013-11-04)

3.2 Simplified API

In this section, we present an approach to configuring analysis components with parameters representing complex objects, based on the strategy design pattern. This allows extracting key behavioral aspects of an analysis component into a pluggable object, thus making the component's behavior customizable and improving composability and separation of concerns.

We investigate if and how current processing frameworks support this design pattern - apart from the analysis components themselves - and find that the support of complex objects as component parameters is largely limited to specific kinds of objects, e.g. providing an abstraction over linguistic resources, such as dictionaries. Then, we present an extension to the Apache UIMA [10] framework which improves the support of the strategy pattern. Finally, we discuss two use cases to illustrate the usefulness of this design for modeling analysis workflows.

These contributions address the following issues in our overall scenario (Figure 3.7):

❷ **Assembling automatic analysis components into workflows is too complex.**
The behavior of analysis components can be configured via strategies. This allows a user to customize the behavior of an analysis component without the need for programming. It reduces the skills necessary to act in the *workflow assembler* role.

❽ **Implementing new interoperable automatic analysis components is too complex.**
The ability to extract arbitrary behavioral aspects into strategies opens up a completely new way to design analysis components. It can provide the *analysis developer* with a more natural and convenient way to implement analysis components.

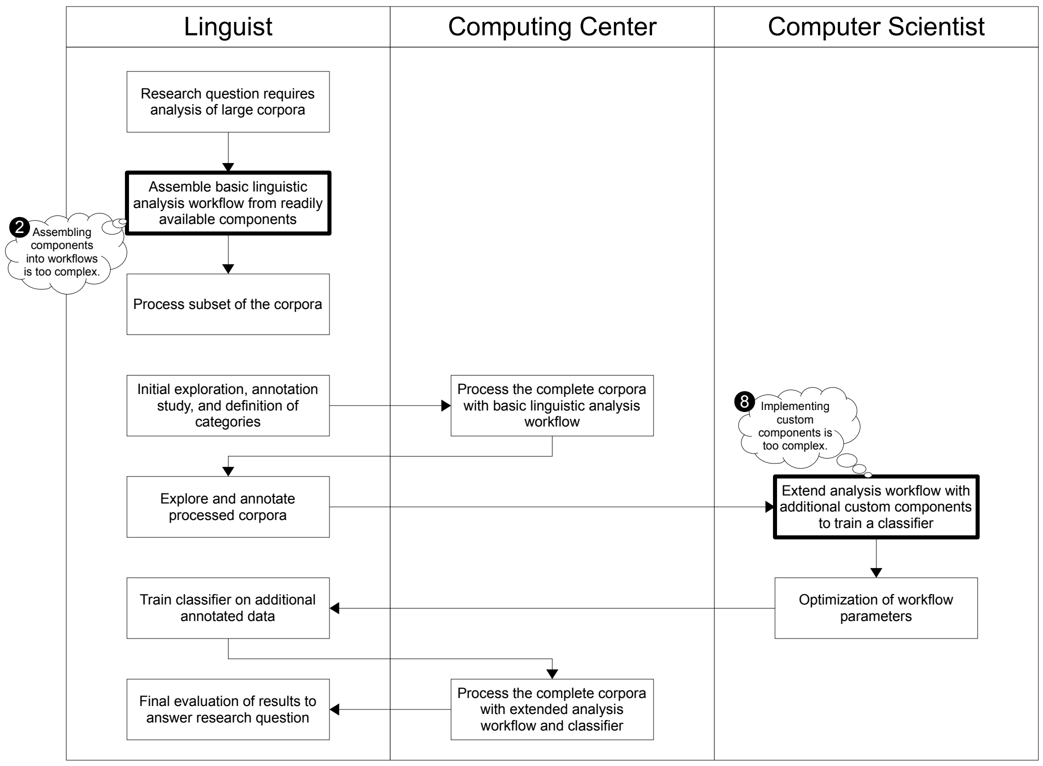

Linguist
Computing Center
Computer Scientist
Research question requires analysis of large corpora
Assemble basic linguistic analysis workflow from readily available components
❷ Assembling components into workflows is too complex.
Process subset of the corpora
Initial exploration, annotation study, and definition of categories
Process the complete corpora with basic linguistic analysis workflow
❽ Implementing custom components is too complex.
Explore and annotate processed corpora
Extend analysis workflow with additional custom components to train a classifier
Train classifier on additional annotated data
Optimization of workflow parameters
Final evaluation of results to answer research question
Process the complete corpora with extended analysis workflow and classifier

Figure 3.7: Issues addressed by our improved support for the strategy pattern in the parametrization of analysis components

3.2.1 Motivation

Processing frameworks allow the composition of analysis components into analysis workflows. For the analysis components to be reusable effectively, they must be configurable, so that they can be contextualized when a workflow is assembled for a particular kind of analysis.

The concept of an *analysis component* itself is an instance of the *strategy pattern* [96]. This pattern occurs if some *context* (e.g. an *analysis workflow*) can be customized via different *strategy implementations* (e.g. a *part-of-speech tagging component* or a *parser component*) which implement a *strategy interface* (e.g. *analysis component*) defining the interaction between the context and the strategy implementations (Figure 3.8).

A good support for the strategy pattern by allowing analysis components to be configured with complex objects which are configurable themselves, can provide a more declarative and intuitive approach to the assembly of an analysis workflow. A set of strategies may already be provided by an analysis component and be readily used by a non-expert programmer. More skilled programmers may conveniently realize new strategies by implementing the task focused strategy interfaces in own classes. Modifying the analysis component or inheriting from it are not necessary to customize key behavioral aspects. Therefore, this approach improves the usability aspect of automatic analysis components for both, the non-expert programmer and the expert programmer.

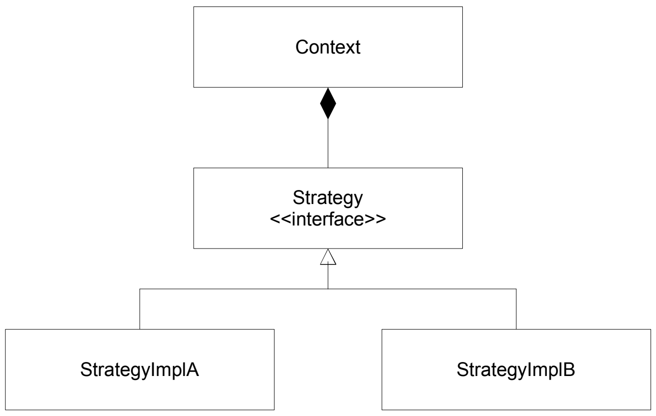


Figure 3.8: *Strategy pattern* of software design

The ability to configure analysis components tends to be limited (Figure 3.9). E.g. a parameter may only assume simple values or lists of such values, like strings, numbers, etc. In addition to the parameters, an analysis component can only use its input data to control its behavior, i.e. the primary data and the analysis results of previous components.

Beyond simple parameters, a processing framework may support some kind of *resource* abstraction, which tends to target in particular the abstraction of sources of data, such as dictionaries, databases, index, etc. Hence, the strategy interfaces used for these resources tend to define methods to interact with these resources, just as the strategy interfaces for analysis components define methods to analyze the data. They are not meant to be used generically to provide analysis components with arbitrary kinds of strategies.

There are, however, cases in which the ability to use the strategy pattern for the configuration of analysis components is beneficial for a usable design.

3.2.2 State of the art

The ability to configure analysis components with complex objects to influence their behavior is apparently useful and can provide an improved usability. We will briefly describe different

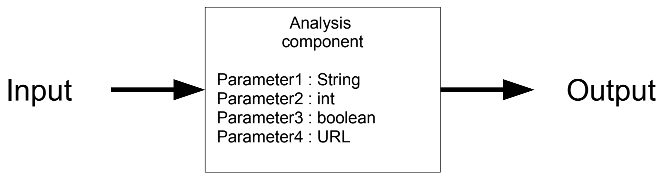


Figure 3.9: Basic structure of an analysis component

approaches to make the behavior of analysis components customizable, including the strategy pattern. Then, we examine the state-of-the-art processing frameworks to determine if they support any of them, which, and how. We observe that these frameworks offer surprisingly little support to configure analysis components for different behaviors, and that even if there is support within the framework, the strategy pattern is hardly used.

3.2.2.1 Mode selection

A set of well-known parameter values, each an identifier for a particular behavior, can be used to configure an analysis component to behave in a certain way. Consider an analysis component to calculate the average length of a word in a corpus with a parameter *mode*. We can set *mode* to the value *mean* to calculate the average as the arithmetic mean, or e.g. to *median*. In any case, the component only supports a certain predefined set of modes to calculate the average word length.

3.2.2.2 Inheritance

An alternative approach would be the use of inheritance. For example, in a machine learning scenario, the training component could be an abstract class. From this, the user could derive a custom class which sets up the feature extractors as part of its initialization routine. This, however, would require the user to actually program. Such an approach is currently being taken by the ClearTK machine learning framework [172].

3.2.2.3 Strategy pattern with internal instantiation

The name of a class or a list of class names is passed to a component. All the classes are expected to implement a certain interface, so that the analysis component may internally create instances of these classes and invoke certain methods. Consider the component for calculating the average word length again. This time, the parameter *mode* accepts the name of a class implementing the following interface:

Listing 3.6: Interface for strategies of calculating the average word length of a list of words

```
interface AverageLengthFunction {
  int averageLength(String... words);
}
```

The framework may already come with several implementations of this interface, e.g. *ArithmeticMean* and *Median*. We can pass one of these or we can choose to implement our own class, e.g. *Maximum*, which would return the length of the longest word in the corpus. In this approach, there is no predefined set of behaviors. Any user can provide a custom behavior creating a new class implementing the *AverageFunction* interface.

Just as analysis components can have parameters, it is sometimes necessary that individual strategies also have parameters themselves. For example, a feature extraction strategy which generates a feature from a sliding window of tokens may have a parameter to determine the window size. However, in the present approach, there is no way to directly configure the strategy with this parameter, because only the class name of the strategy is passed to the analysis component, not a readily configured strategy instance.

For this reason, the approach is often combined with a concept that we call *parameter forwarding*. Instead of setting the strategy parameters on each strategy individually, they are set on the analysis component to which the strategy is passed. The component is then responsible to forward the parameters to the strategy when it is instantiated internally by the component.

This approach requires special conventions to ensure that parameters reach the strategy that they are meant for. ClearTK, for example, prefixes each parameter name with the class name of the strategy to which the parameter should be forwarded. This convention does not work when the same strategy is passed multiple times to an analysis components, but with different parameter settings, for example when we want to pass multiple sliding window feature extractors with different window sizes to our machine learning component. More sophisticated naming conventions are conceivable, but put an increasing burden onto the user to follow them.

3.2.2.4 Strategy pattern with external instantiation

Instead of leaving the responsibility to forward parameters to analysis components, the processing framework itself may offer provisions to configure and instantiate strategies before they are passed on to the analysis components. Thus, the strategy is instantiated externally by the processing framework rather than internally by the analysis component.

All the processing frameworks we examine operate based on a *component descriptor* which defines how an analysis component is instantiated, what the parameters are, what values they have, etc. The frameworks need this information so they can create instances of components based on this descriptor whenever and wherever necessary, e.g. on a compute cluster after all required software libraries and resources have been automatically deployed to the cluster nodes. Another reason to operate with component descriptors, instead of using regular code to instantiate components, is the support of graphical tools for configuring components and assembling workflows. It is pretty straightforward to implement such tools operating on a declarative model of the component or workflow.

Thus, just as an analysis component is configured by a descriptor, a strategy also needs to be configurable using its own descriptor, which includes all the parameter settings for the strategy. In this way, we can also configure a strategy several times, using several descriptors, each with its own set of parameters. This makes it easy to configure multiple sliding window feature extractors with different window sizes on a machine learning component.

3.2.2.5 Support in processing frameworks

Next, we examine the processing frameworks to determine if and how they are addressing the need for strategies.

GATE

GATE appears to be the most flexible of the examined processing frameworks when it comes to the configuration of components. In addition to the usual simple values, any type of *GATE resource* can be used as a parameter value and any class following the Java Bean conventions can be used as such resources. According to these conventions, the bean class must provide a no-argument constructor and provide getter and setter methods for any of its configurable properties. Examining the source code of GATE, however, it appears that the capability of configuring components with complex objects is used mostly for *LanguageResources* (LRs), which represent documents, corpora or other sources of data. In some cases, enumeration types, representing a fixed inventory of strategies, are used as parameters. In a few cases, class names are used as parameters, e.g. to allow using custom operators in JAPE scripts. We did not find a case where a complex object other than a data source was used as a configuration value. Possibly, this is an effect of the way the graphical user interface of GATE is designed. It allows creating and managing four types of resources: *applications* (analysis workflows), *language resources* (data sources), *processing resources* (analysis components), and *data stores* (storage for analysis results). Strategies such as the feature extractors, splitters or rankers from our example simply do not fit into this user interface.

Tesla

Tesla appears to be the least flexible framework when it comes to the configuration of components. It only supports the typical primitive Java types[8] as well as lists of these. This is interesting to note, since otherwise Tesla pays much attention to exploiting the object-oriented features of the Java language. The limitation to these primitive types is slightly alleviated by the fact that custom editors can be registered for each parameter. A String parameter representing a file name may be editable in the user interface using a file chooser. If the parameter may assume any of a predefined set of values, only these values are offered in the user interface. A special enumeration data type is not used. A String parameter representing a class name may be editable via a visual component allowing the selection of one of the implementations of a certain interface.

In a situation where a component is configured with a strategy that itself needs to be configured with parameters, Tesla components appear to simply declare additional parameters on the component. For example, the *Suffix Select* component has two String parameters accepting names of classes, *selectAlgorithm* and *weightCalculatorClass*. The component creates instances of these two classes and then invokes a setter method on the algorithm instance to which it passes the weight calculator instance. This is obviously not a generic solution. It would not be possible to implement a new custom selection strategy which requires additional parameters.

UIMA

The UIMA framework is situated somewhere in between the previous two frameworks with respect to the configuration of analysis components. Parameter values in UIMA are even more restricted than in Tesla. Only a few primitive types (*integer*, *boolean*, *float*, and *String*) and lists of these are natively supported. Via the uimaFIT API, many other simple value types (e.g. *File*, *URL*, ...) are also supported. These are automatically coerced into a String representation when passed into a component and transformed to the actual parameter type when the analysis component is initialized. Additionally, UIMA has a concept of so-called *shared resources*. UIMA provides several interfaces based on which shared resources can be implemented:

- **DataResource** – an interface for accessing data resources from an URL or stream. The resource is created and initialized when the workflow is instantiated from its descriptor.
- **ParameterizedDataResource** – an interface for accessing data resources which require runtime information, such as the language of the document being processed, and, thus, cannot be fully initialized when the workflow is created.
- **Resource** – a generic interface which is implemented by most components in the UIMA framework including analysis components. It allows creating custom resources[9] which are not specialized for data access, e.g. for implementing arbitrary strategies.

[8] Primitive Java types: *long*, *int*, *short*, *boolean*, *character*, *double*, *float*, and *String*

[9] Due to the way the descriptors for these resources have to be written, using the UIMA *CustomResourceSpecifier* class, parameter can even be only of the type *String* – they cannot even be multi-valued.

The shared resources in UIMA are mostly used to implement different strategies of data access. The ClearTK machine learning framework [172], which is based on UIMA, could make use of external resources to model feature extraction strategies, but instead, a combination of inheritance and parameter forwarding is used.

3.2.3 Contribution: Improved support for configurable analysis component behavior

In this section, we present an improved approach to support configurable analysis component behavior based on external instantiation. Our approach allows the flexible combination of strategies with analysis components in such a way, that each strategy can be individually configured with parameters or other strategies. This allows a clear separation of concerns between the strategies and the analysis components. Details of our concept are specific to the UIMA framework, because we implemented our concept as part of the uimaFIT library. However, the motivation for using strategies to configure component behavior is independent of the processing frameworks, and none of the frameworks supports this approach sufficiently, as we previously discussed in Section 3.2.2.

Our approach to the strategy pattern relies on the UIMA *shared resource* mechanism. In order to avoid naming inconsistencies with the UIMA and uimaFIT libraries and code examples, we use the term *shared resource* as a technical synonym for *strategy* in the rest of this section.

We implemented our concept as an extension to the uimaFIT library, which provides a more usable API for the UIMA framework and fits in well with our idea of a domain specific language for analysis workflow assembly. Adding our extensions to uimaFIT instead of adding them to the core UIMA framework allowed us to publish our work and test our new concepts early without creating a fork of the UIMA framework. The extensions were done in three phases:

- **Injection of shared resources into analysis component** – Adding support to uimaFIT for handling shared resources, creating the typical Java annotations, and factory methods.
- **Injection of nested resources** – Adding the ability to configure shared resources with other shared resources to create a *nesting* of resources required additional effort. The concept of nested resources has so far not been supported by the UIMA framework. Nested resources provide for the composability of strategies.
- **Injection of multi-valued shared resources** – Adding the concept of multi-valued shared resources. UIMA supports multi-valued parameters, but shared resources are always expected to be singular. Having multi-valued shared resources, e.g. to configure multiple feature extractors on a machine learning component, is an important feature. This concept improves the composability of strategies.

In the following illustrative code examples, we do not use the original UIMA/uimaFIT class and method names, which are slightly more verbose.[10] Instead, we use aliases which are more consistent with the names used elsewhere in this thesis. We also ignore the fact that it would actually be necessary to inherit from specific UIMA/uimaFIT classes, and claim that just implementing a certain interface would be sufficient to make a class a valid analysis component or resource.[11]

[10] Actual names in uimaFIT:
`analysisComponent(...)` ↦ `createPrimitiveDescription(...);`
`resource(...)` ↦ `createExternalResourceDescription(...)`

[11] Instead of just implementing the `Resource` interface, it would be necessary to extend the uimaFIT `Resource_ImplBase` class. Also, extending `JCasAnnotation_ImplBase` would be necessary instead of just implementing `AnalysisComponent`.

To give an example of how uimaFIT works, we illustrate the parameter injection support. uimaFIT allows annotating class fields with the `@ConfigurationParameter` annotation (Listing 3.7) and automatically injects parameter values into these fields when a component is initialized. When configuring a component within an analysis workflow, the component class name is followed by a list of name/value pairs representing the configuration parameters and their values (Listing 3.8). This saves considerable boiler-plate code, otherwise necessary to manually extract a parameter value from the configuration context of the analysis component and to set up a valid component descriptor.

Listing 3.7: Injection of a parameter into an analysis component – parameter declaration

```
class WordFilter implements AnalysisComponent {
  @ConfigurationParameter
  int minLength;
}
```

Listing 3.8: Injection of a parameter into an analysis component – setting the parameter

```
analysisComponent(WordFilter.class,
  "minLength", 3);
```

3.2.3.1 Injection of shared resources into analysis components

The support for injecting shared resources into analysis components has been implemented such that is appears very similar to the already existing support for parameter injection.

Listing 3.9: Injection of a shared resource into an analysis component – declaring the resource dependency

```
class Learner implements AnalysisComponent {
  @ExternalResource
  FeatureExtractor featureExtractor;
}
```

A new Java annotation `@ExternalResource` (Listing 3.9) has been added, which can be used to annotate class fields and to allow uimaFIT injecting shared resources into these fields. This injection is handled by a new helper method provided by `ExternalResourceInitializer`, `initialize(component, context)`, which operates as an adapter between the analysis component's configuration context and the component itself. The helper method is invoked during the initialization of an analysis component.

Listing 3.10: Injection of a shared resource into an analysis component – setting the resource

```
analysisComponent(Learner.class,
  "featureExtractor", resource(NGramFeatureExtractor.class, "nGramSize", 3));
```

The uimaFIT factory methods for the creation of analysis component descriptors have been extended, so that shared resources can be configured similar to normal configuration parameters (Listing 3.10). It needs to be noted, though, that this convenient way of configuring an analysis component with a shared resource is only possible if the analysis component has been built using the uimaFIT annotations [12].

3.2.3.2 Injection of nested resources

The UIMA framework allows declaring that an analysis component requires some shared resources, but it provides no means of declaring that a shared resource requires another shared resource. Therefore, supporting the injection of shared resources into other shared resources (*nested resources*) was more difficult to solve.

From the user's perspective, it should be a natural extension of the shared resource support in uimaFIT. Adding a field in a shared resource class and marking it with the `@ExternalResource` annotation should be sufficient (Listing 3.11). Likewise, the assembly of an analysis workflow containing a shared resource which is configured with another shared resource should work without any additional effort (Listing 3.12).

Listing 3.11: Injection of a shared resource into another shared resource – declaration

```
class Learner implements AnalysisComponent {
  @ExternalResource
  FeatureExtractor featureExtractor;
}

class SimilarityFeatureExtractor implements Resource, FeatureExtractor {
  @ExternalResource
  SimilarityMeasure similarityMeasure;
}

class MySimilarityMeasure implements Resource, SimilarityMeasure {
}
```

Listing 3.12: Injection of a shared resource into another shared resource – injection

```
analysisComponent(Learner.class,
  "featureExtractor", resource(SimilarityFeatureExtractor.class,
    "similarityMeasure", resource(MySimiliarityMeasure.class)));
```

What appears simple from the user's perspective, turned out to be rather difficult to implement, due to the way that the UIMA framework handles shared resources:

- **Binding** – The UIMA framework allows declaring that an analysis component requires some shared resources. However, it provides no means of declaring that a shared resource depends on another shared resource. In order to model this information, uimaFIT now uses a customized version of the UIMA `ExternalResourceDescription` implementation, which allows making such declarations. This information needs to be retained only while interpreting the analysis workflow assembly, between the time that a shared resource descriptor is created and the time that bindings between shared resource and analysis components are generated. After the binding is complete, the information can be safely discarded. Hence, further extensions or changes to the UIMA framework, e.g. of the XML representation of analysis component descriptor or analysis workflows, were not required. Nested resource bindings are registered on the component to which the resource is bound transitively. To disambiguate the keys to which the resources are bound, keys for nested resources are internally prefixed with the unique name of the resource. That way, keys with the same name in the host component or any of the resources, are unique and do not conflict.

- **Initialization** – Shared resources, with an exception of the parametrized data resources, are initialized when an analysis workflow is instantiated. During this process, the *initialize* lifecycle event is triggered immediately after the shared resource has been instantiated. At this point, other shared resources may not have been instantiated yet and thus cannot be fetched and injected into the annotated class fields. For this reason, uimaFIT introduces a new *after resources initialized* lifecycle event for shared resources. This is triggered when the `ExternalResourceInitializer` is used for the first time to inject shared resources into an analysis component. At the time the analysis components are initialized, all possible shared resources have already been instantiated. The *after resources initialized* event is guaranteed to be triggered all resources have been instantiated and before the workflow starts processing data. Eventually, it would probably be better to remove this extra lifecyle event and defer the *initialize* event on shared resources until after all resources have been instantiated. That, however, would have required invasive changes to the core UIMA framework, which was to be avoided.

3.2.3.3 Injection of multi-valued shared resources

Some scenarios require that multiple strategies can be passed to a component. In the text classification scenario, we want to pass multiple feature extractors to the learner component. The UIMA framework does not allow the binding of more than one shared resource to a given dependency key of a component.

From the perspective of a developer implementing a component, adding support for multi-valued shared resources in a component or resource is natural, as the annotated field is changed to an array or a Java collection type (Listing 3.13). From the perspective of the user of a component, passing a multi-valued resource as a configuration parameter is likewise straightforward, by passing an array or collection of resource descriptors (Listing 3.14).

Listing 3.13: Injection of a multi-valued shared resource into an analysis component

```
class Learner implements AnalysisComponent {
  @ExternalResource
  FeatureExtractor[] featureExtractor;
}
```

Listing 3.14: Injection of a multi-valued shared resource into an analysis component

```
analysisComponent(Learner.class,
  "featureExtractor", list(
    resource(AverageWordLengthFeatureExtractor.class)
    resource(NGramFeatureExtractor.class, "nGramSize", 3)
    resource(NGramFeatureExtractor.class, "nGramSize", 2)
    resource(NGramFeatureExtractor.class, "nGramSize", 1));
```

This functionality was implemented leveraging the *nested resources* support of uimaFIT. A new special-purpose resource called `ResourceList` was defined. This resource declares a single configuration parameter, namely the number of resources in the list. Otherwise, it serves as an anchor to which the individual resources in the list (the *list elements*) are bound. Whenever a multi-valued shared resource with n values is passed, uimaFIT internally creates such a `ResourceList`, dynamically generates n resource dependencies with the keys *ELEMENT[0]* to *ELEMENT[n-1]* and binds the resources to these.

To properly initialize the class fields into which such multi-valued resources are injected, the `ExternalResourceInitializer` was extended. It reads the size of the list from the *ResourceList* resource and fetches all individual list elements, and adds them to a Java collection `List` with which the field is initialized.

3.2.4 Examples

We employed the improved support for the strategy pattern in two use-cases: *decompounding* of compound words and *feature extraction* in a workflow used to train a classifier. These are illustrated in the following sections.

3.2.4.1 Scenario 1: Decompounding

Decompounding is the task of separating a compound word into smaller semantic units. For example, the word *Fließbandarbeit* (*assembly-line work*) is a compound of the three words *fließen* (*run*), *Band* (*belt*), and *Arbeit* (*work*). It is also a compound of the words *Fließband* (*assembly-line*) and *Arbeit* or maybe *fließen* and *Bandarbeit*.

Workflow structure

There are two sub-tasks involved in decompounding. The first task is to reverse the compounding step. *Splitting* is the identification of locations within the compound word, at which the word may be split, preferable into those components that were originally joined to form the compound. Obviously, the compound may not just be split at any location and should yield a semantically plausible decomposition. The second task, *ranking*, is deciding the order in which the splits should be made. For the word *Fließbandarbeit*, the correct splits and the splitting order is *((Fließ|band)arbeit)*, while *(Fließ(band|arbeit))* would be wrong.

There are different strategies for both tasks, for example:

- **Splitting**
  - **Left-to-right** – Scanning a word from left to right, splitting off the prefix whenever it corresponds to a word in a dictionary.
  - **Data-driven** [136] – For each position within a word, the number of words in dictionary is determined, that contain the result of a split operation at this position as a prefix (w_{prefix}) or suffix (w_{suffix}). A split is then made based on where the biggest difference in the number of words is found ($max\ |w_{prefix} - w_{suffix}|$).
- **Ranking**
  - **Compound probability** [4] – A word is split into compounds such that the sum of weights of each compound part is minimized. The weight is calculated based on the frequency a compound part occurs as a separate word in a large corpus.
  - **Mutual information** [4] – A word is split into compounds such that the mutual information score is maximized. This score is calculated by looking at how often the parts occur *near* each other in a large corpus.

A traditional analysis workflow for decompounding might follow the design depicted in Figure 3.10. The *splitter* component determines all possible splits for each word and generates annotations on the document text for each of them. These annotations are then read by the *ranker* component which assigns a rank to each split for each word and removes all but the best-ranked split annotations. To vary the splitting and ranking strategies, different analysis components are implemented, each using a specific splitting or ranking strategy.

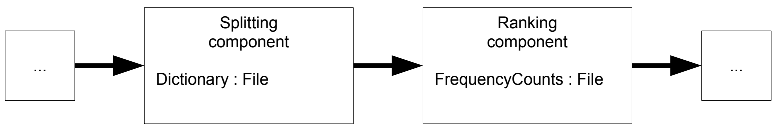


Figure 3.10: Traditional decompounding workflow using *splitter* and *ranker* components

This approach has several drawbacks.

First, the user wants to perform a single task, namely decompounding, but has to construct a workflow in which this task is actually rendered as two tasks. This is not very intuitive for a user who is mainly interested in a good result and not necessarily in the way how exactly this result was achieved.

Second, a potentially very large amount of splitting information needs to be rendered as annotations, so that the information can be passed on to the ranker. This appears unnecessary, since the user in the end is mostly interested in only the best decompounding for each word. Hence, the splitting information is only required in the communication between splitter and ranker. It is convenient to use lightweight data objects instead of having to render this information as data exchange objects of the processing framework. Finally, the splitting and ranking strategies, again, may be configured with dictionaries or word frequency lists. We can easily imagine that these again come in different formats, so that encapsulating them as shared resources with a uniform API is useful to make them easily reusable by the splitting/ranking strategies or other analysis components requiring such kinds of resources.

Using strategies as parameters

With a processing framework supporting the strategy pattern for the configuration of analysis components, a different design can be used (Figure 3.11). A single decompounding component is added to the analysis workflow. This component is then configured with a splitting and a ranking strategy, both complex objects which are configured themselves. In the given example, the splitting strategy is configured with a dictionary resource and the ranking strategy is configured with a word frequency count resource. Both of these resources are again complex objects which may come in different flavors, e.g. loading their data from files, from a database, or from an online service.

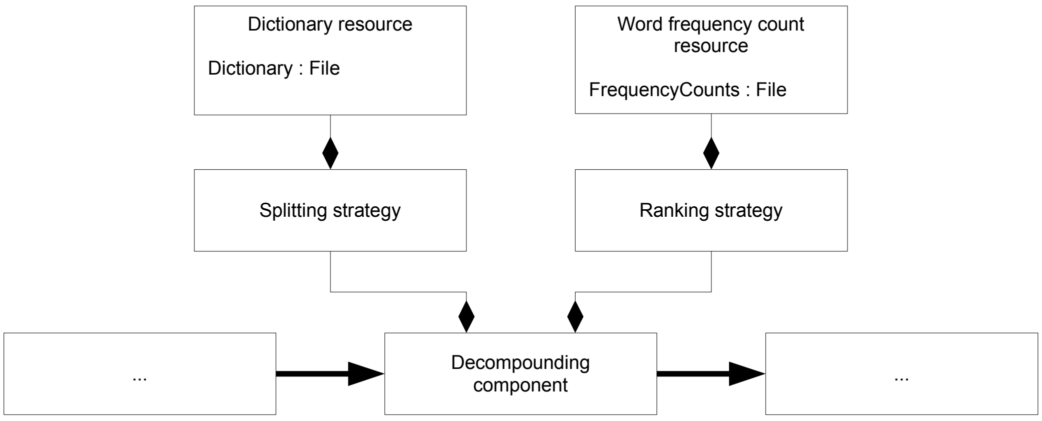


Figure 3.11: Decompounding workflow using a single decompounding component, configured with strategies for splitting and ranking

This approach to modeling different splitting and ranking strategies has been implemented in the decompounding module of the *DKPro Core* component collection (Section 5.2).

3.2.4.2 Scenario 2: Text classification

Text classification is the task of training a classifier by learning characteristic features from a body of data and applying this knowledge to characterize unseen data. An application of text classification is, for example, learning how to separate spam mails from non-spam mails. We examine the task up to the generation of the classifier. How to apply the classifier to unseen data does not contribute any additional insights with regards to the design of the software architecture.

Workflow structure

A traditional analysis workflow (Figure 3.12) for training a classifier may consist of a series of feature extraction components, each reading results of some previously run preprocessing analysis and rendering extracted features, such as average word length, part-of-speech n-grams, etc., as annotations according to the standards of the processing framework.

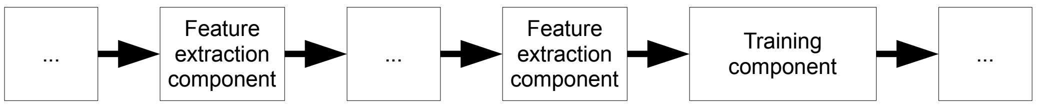


Figure 3.12: Traditional training workflow using several *feature extractor* components, and a *training* component

Again, a different design (Figure 3.13) can be used with a processing framework supporting complex objects as component parameters. This scenario is different from decompounding, as not a single, but a list of feature extraction strategies is passed to the training component.

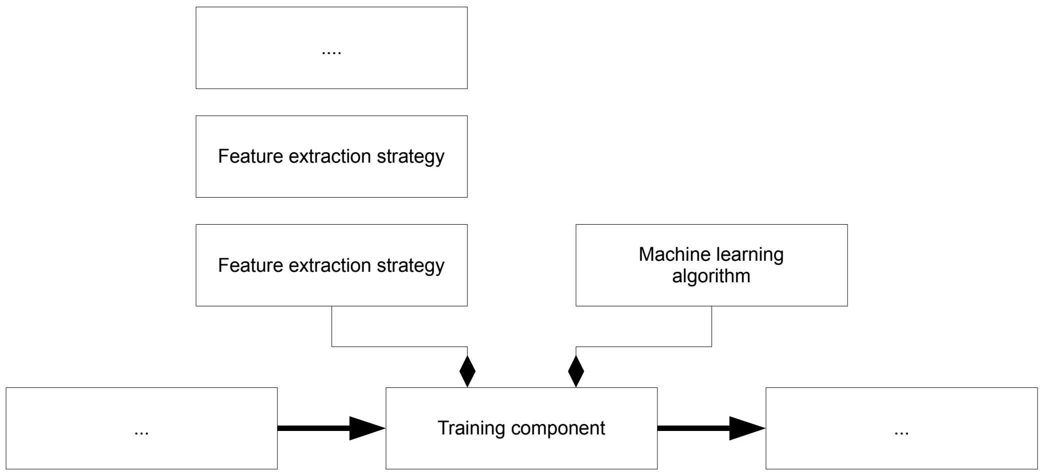

....
Feature extraction strategy
Feature extraction strategy
Machine learning algorithm
...
Training component
...

Figure 3.13: Training workflow with a training component that can be configured with a list of feature extraction strategies and a learning algorithm

Using strategies as parameters

The benefits of the strategy-based design over the traditional design are again, that extracted features need not be rendered as framework data objects. Instead, they can be directly passed from the feature extractors to the machine learning algorithm within the training component. Additionally, feature extractors can be easily parametrized individually. This promotes the development of generic, configurable feature extractors which can easily be added to the training component in different configurations (cf. 3.14). Such an approach to configuring a machine learning component with feature extraction strategies has been implemented in the DKPro TC [65] framework.

3.2.5 Summary

In this section, we have discussed that the strategy pattern is a useful means to provide users with a new level of flexibility for the configuration of the behavior of analysis components without requiring them to program. We found that existing processing frameworks offer only limited support for strategies, e.g. to provide an abstraction over linguistic resources, such as dictionaries. In particular behavioral strategies, as opposed to resource access strategies, are not well supported.

We designed and implemented an extension to the uimaFIT library for the UIMA processing framework, which facilitates the implementation of strategies based on the UIMA shared resources API. The uimaFIT library was extended to support UIMA shared resources. Beyond that, our extensions allow the strategies to be configured with strategies themselves and the ability to pass multiple strategies of the same role to a component, e.g. multiple feature extractors to a machine learning component. The latter two concepts are currently not supported by the UIMA framework itself. We integrated our extension into the official uimaFIT codebase.

In the DKPro Core component collection (Section 5.2), we apply our approach in the context of decompounding. The DKPro TC [65] framework employs it to allow the user to flexibly configure and compose feature extractors. Also, the ability to customize behavior can be improved, e.g. by defining new lightweight behaviors as part of an analysis workflow, without the need to implement a new analysis component from scratch.

In future work, we plan to extend our approach with a concept for default strategies. While most processing frameworks offer the ability of defining default values for parameters, we did not define a corresponding concept of defining default strategies as part of our current contribution. Default values for parameters can typically be discovered by inspecting an analysis component programmatically, e.g. by analyzing the metadata encoded in Java annotations. Likewise, a solution for default strategies should allow the default strategy configuration to be inspected. This allows automatically extracting information about the default strategy from the analysis component and recording it in documentation or in component metadata, where it can be used to discover components. Therefore, a trivial solution, in which a component internally sets up and uses a default strategy unless another strategy has been explicitly configured, is not sufficient. Such internal behavior of the component cannot be easily inspected from the outside.

4 Reproducibility

As it has been stated by many researchers, *reproducibility is the hallmark of good science*. Building on the work by others, commonly referred to as *standing on the shoulders of giants* is central to science. Traditionally, scientific work is published in the form of conference papers or journal articles, but these media are unable to capture the intricate details of the actual implementation of an experimental setup [206]. A growing *reproducible research* movement, as Drummond [68] calls it, can be observed in various scientific domains, particularly in those relying on computational simulations.

Drummond [68; 67] reviews several sources discussing the topic of reproducible research and comes to the conclusion that this topic actually consists of three sub-topics:

- **Reproducibility** – the aim of reproducibility is to provide the ability to exactly repeat an experiment at a different time, in a different environment or by a different person, producing exactly the same results. Repeating an experiment exactly means, that neither the experimental workflow nor the data is subject to variation. The goal is to provide means of locating details in the experimental workflow which have significant impact – positive or negative – on the results, but may not have been sufficiently explained along with the published results.

- **Statistical replicability** – the aim of statistically replicating an experiment is to generalize over a specific implementation of the experiment, the data set, or other resources involved in the experiment. These may imply a strong and undesired bias on the experimental results.

- **Scientific replicability** – this aims at the verification of the actual abstract hypothesis underlying a series of potentially completely different experiments, approaching a phenomenon from a variety of different angles.

We do not enter here into the controversy regarding the motivations underlying the growing desire for reproducibility. The fact is that various approaches are taken towards improving the reproducibility, from service infrastructures that allow maintaining and running experiments in the cloud [116] to capturing the execution environment and result log of experiments [55]. While these approaches have certain merits, we chose to approach the problem from a different angle, aiming to preempt the need of capturing the execution environment and allowing the user to keep control over running an experiment and preserving it in a runnable state.

In Section 4.1, we discuss the portability of analysis workflows. In order to achieve reproducibility, it is an essential prerequisite that a workflow functions and produces the same results in different environments. Therefore, it is mandatory that the workflow is *portable*, meaning that it can be moved from one environment to another, including all required software, primary data, and other relevant resources. Instead of capturing the workflow environment, we suggest that a workflow should inherently provide sufficient information to create its environment. Based, among other techniques, on the automatic resource selection (Section 3.1) and on the collection of analysis components (Section 5.2), we demonstrate how such portable workflows can be implemented conveniently. We consider it most important to maintain usability while striving for reproducibility, since there can be little hope for acceptance otherwise.

In Section 4.2, we discuss how to eliminate spurious manual interventions from automatizable workflows. Manual steps are an inherent source of error. Yet, analysis workflows tend to contain steps which are performed manually, because the processing framework being used does not allow conveniently automatizing these steps. We introduce a framework-independent approach to building dynamic, data-flow oriented workflows, particularly targeted at facilitating parameter sweeping experiments involving multiple interacting analysis workflows. Parameter sweeping can be used to find the parametrization of an experiment producing the best results. In dynamic workflows, the assembly of the analysis workflow itself becomes subject to parametrization. As such, dynamic workflows are well suited to build an aggregate experiment setup, testing the statistical replicability of an experiment by varying essential parts, from the data set to the implementation of analysis components.

4.1 Portable workflows

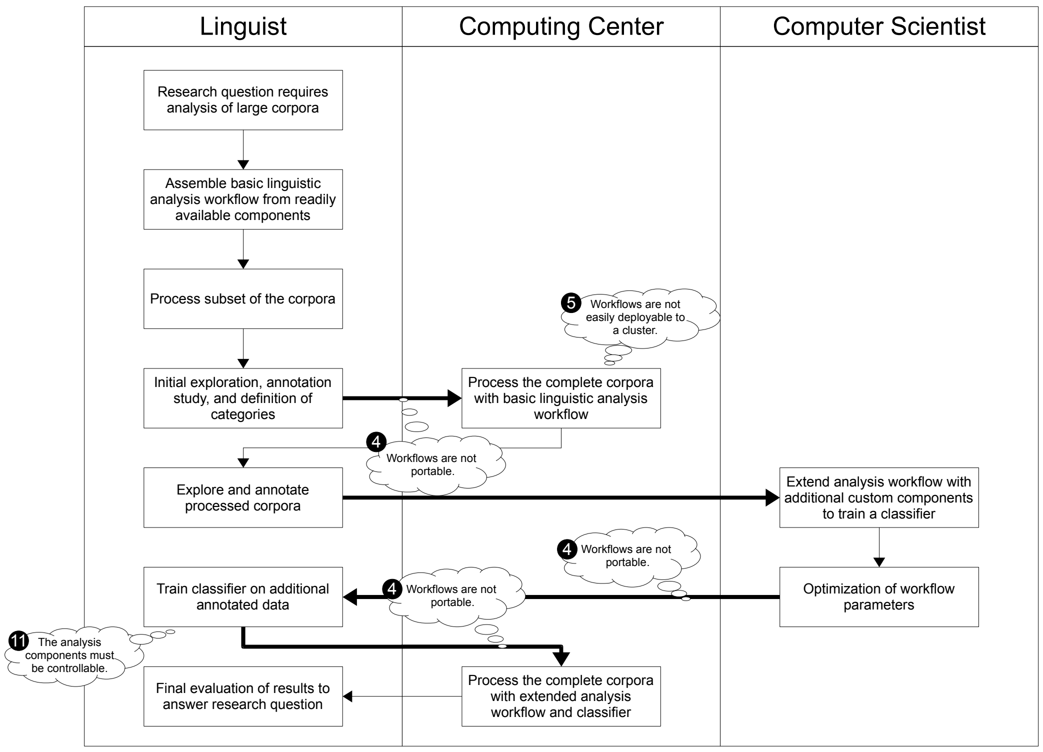
In this section, we propose an approach to portable, reproducible analysis workflows. We discuss how analysis workflows can conveniently retrieve portable software from repositories and what benefits and issues such an approach entails. We compare this to alternate approaches that aim at facilitating reproducible research and how these approaches are supported by current processing frameworks. We note that current processing frameworks allow building analysis workflows and sharing them, but leave all responsibility of setting up the execution environment to the user. To address this, we present an approach which encodes the high-level logic of the experiment in a script containing sufficient information to provision the experiment including its dependencies to an environment in which it can be run.

Our proposal addresses the following issues in our overall scenario (Figure 4.1):

❹ **Workflows are not portable between computers.**
Our approach to analysis workflows based on portable software allows the workflows to be easily exchanged between researchers and to reproduce each others results.

❺ **Workflows are not easily deployable to a compute cluster.**
Portable analysis workflows which reference specific versions of components and resources can be deployed to a compute cluster more easily.

⓫ **The user has no control over workflows that rely on expert skills from a different domain, undocumented knowledge, or third-party infrastructures, e.g. web services.**
Portable analysis workflows put the users in control by allowing them to maintain usable copies of all involved artifacts. They do not have to rely on third-party services or partners.



Linguist
Computing Center
Computer Scientist
Research question requires analysis of large corpora
Assemble basic linguistic analysis workflow from readily available components
Process subset of the corpora
Initial exploration, annotation study, and definition of categories
Explore and annotate processed corpora
Train classifier on additional annotated data
Final evaluation of results to answer research question
Process the complete corpora with basic linguistic analysis workflow
Process the complete corpora with extended analysis workflow and classifier
Extend analysis workflow with additional custom components to train a classifier
Optimization of workflow parameters
5 Workflows are not easily deployable to a cluster.
4 Workflows are not portable.
4 Workflows are not portable.
4 Workflows are not portable.
11 The analysis components must be controllable.

Figure 4.1: Applicability of portable workflows within the motivating scenario

4.1.1 Motivation

Research results, as published in conference papers or journal articles, are unable to capture the details of the actual implementation of an experimental setup [206]. Not at least for this reason, there is a desire to provide researchers with a convenient way of rerunning an experiment workflow and to reproduce the published results. But there are other motivations as well:

- **Cooperation** – As illustrated in our initial scenario, portable workflows are essential for cooperating researchers, even before any results are published. The cooperation partners need to be able to run the experiment and produce the same results independently of each other, so that each partner can work on improving certain aspects of the experiment.
- **Reuse** – Workflows should remain usable by each partner after the cooperation comes to an end. Our initial scenario highlights a cooperation between programming experts and non-programming experts. In such a scenario, it is particularly important, that the non-expert programmer can continue using the jointly built workflow without the assistance and resources from the programming-expert partner. Portable workflows permit the cooperation partners to preserve all data and software required by the workflow without relying on partners or other third parties.
- **Scaling** – A simple workstation may only be sufficient to run an experiment on a limited data set. In order to operate on more data, it may be necessary to run an experiment on a powerful server or even a compute cluster, possibly running on a different hardware architecture and operating system. Without a workflow relying on portable software and data, this would hardly be manageable.
- **Debugging** – Debugging analysis components or an experiment workflow running on a compute cluster is hardly trivial. On a cluster, an experiment is spread over multiple machines and multiple processes, which makes it difficult for a debugger to monitor the overall system state, to interrupt processes when a breakpoint is reached, and to step through the program. The task is significantly easier on a local workstation, within an integrated development environment. Thus, the ability to debug a component locally and later run the same component with more data on a cluster is important to maintain reasonable turnaround times during development and debugging.

However, there are issues which may cause a published experimental workflow to become non-functional or which may prevent that an experimental workflow is published in its entirety, i.e. including all required software, resources, and primary data:

- **Legal issues** – Among the many legal issues involved in the licensing of research results, research software, and research data [205], we here pick redistributability as particularly crucial to reproducibility. Portability is not only a technical issue, but also a legal issue. If the license under which research software or data is published does not permit redistribution, such work or work building thereupon cannot be freely shared between cooperating researchers and it cannot be distributed via an independent repository. Technical as well as legal portability issues are repeatedly cited as motivations for building walled garden[1] web-service-based infrastructures, for example in the proceedings of a recent workshop on service-oriented architectures [115], at which members of two large infrastructure projects of the digital humanities discussed solutions and impacts of such infrastructures.

[1] The term *walled garden* refers to an ecosystem or platform in which users may participate, even provide content or services, but the platform provider acts as a gatekeeper and has the power to lock users out (even from their own previously supplied content) or to reject content or certain ways of using the system.

- **Workflow decay** – Platforms, such as *myExperiment* [103], provide means of publishing workflows and sharing them with the community. The workflow descriptions shared there, however, are subject to *decay*, meaning that they cannot be used anymore as intended after some time. Hettne et al. [114] conducted a survey of workflows published on myExperiment and studied the causes for decay. They suggest, among other things, that web services are one of the major reasons for workflow decay and suggest that relying on portable (platform independent) code helps to prevent decay.

Trust and control

Gómez-Pérez et al. [104] suggest introducing a measure for the reliability of workflows, which incorporates *trust* towards the workflow authors, but also towards the providers of web services used by the workflow. While we do not build a formal method of measuring workflow quality here, we suggest that, from the individual researcher's point of view, control is better than trust. Whenever trust in a volatile service or data provider can be replaced with portable software or data, which can subsequently be preserved and controlled by the user, control can replace trust. For this reason, processing and data, both have to be portable and, consequently, controllable. The choice when to update a workflow to different data or different dependencies remains largely with the user, and is not dictated by third parties, such as service providers.

When relying on trust cannot be avoided, we prefer to trust in wide-spread and largely domain-independent service providers and technologies. For example, the Java programming language and the Java Virtual Machine are widely used in commercial and non-commercial environments and are supported by a strong user and developer community. For the distribution of software libraries, the Maven [180] repository technology has become the de-facto standard and it is likewise used and supported by a strong community of commercial as well as non-commercial users and developers. This community also supports the Maven Central Repository [149], a federated infrastructure for the distribution of software libraries and resources. For this reason, neither of the two technologies and associated infrastructures are likely to disappear or fall into disrepair in the foreseeable future. They appear more trustworthy and reliable than efforts undertaken by individual research communities.

Experimental workflows

We envision a portable workflow description that is minimal, but contains the top-level workflow logic, so one can easily understand what happens, as well as sufficient detail to provision the runtime environment for the workflow. The provisioning of a portable workflow covers the automatic deployment of all required analysis components, resources, and their dependencies to the environment in which the workflow is to be executed. Only a bootstrapping mechanism should need to be present in the environment, which can be used to trigger the provisioning process.

Such a portable workflow is an alternative to processing infrastructures based on web services. It promotes scaling out processing to compute clusters as the processing can be distributed as desired across the cluster nodes. Unlike when using web services, the user is not restricted to the computing resources offered by third parties. Relying on minimal, and themselves, portable prerequisites, such as a Java Virtual Machine (JVM), provides a convenient and reliable means of running an experimental setup at a later time or in a different environment.

4.1.1.1 Portability

The word *portability* is derived from the Latin word *portare (v.)* (to carry, to take). When we talk about portable workflows, we mean *taking a workflow and all associated software and data* *from one environment and running it in some other environment*. This entails several technical and legal aspects.

Technical aspects

With respect to software, the term *portability* is associated with different concepts (cf. [158]):

- **Source portability** – the portability of software is usually associated with a particular hardware or software platform on which the software can be compiled from its sources and where it can be run. We say, *a software is portable to this platform*.

- **Binary portability** – portable software, or stand-alone software, can easily be run from an external storage device (e.g. an USB stick). It does not require special permissions (e.g. an administrator account) and does not make any persistent modifications to the environment it is run in (e.g. store information in the user's profile).

- **Byte-code portability** – a software is portable across platforms if the same compiled binary can be run on different system platforms. An example are the *universal binaries* of the Mac OS X operating system, which contain compiled code for the PowerPC architecture and the x86 architecture. Another example is software which has been compiled to byte-code, a hardware-independent binary representation which requires a runtime environment such as a Java Virtual Machine. Such software runs on any system platform for which the runtime environment is available.

For the purpose of portable workflows, byte-code-based software provides the most convenient solution. The smaller the required runtime environment is, the less does the workflow execution rely on system libraries and implementation details of the runtime environment, the more logic is encoded in portable byte-code.

Unfortunately, not all software is byte-code-based. In that case, a workflow should at least rely on binary-portable software. Such software should depend as little as possible on system libraries. If the software requires additional libraries, these should be statically linked into the executable file. If this is not possible, the libraries should at least be packaged together with the executable file, so that they can be automatically provisioned to the execution environment and the responsibility of installing them does not fall to the user (cf. [45]).

Software which cannot be packaged in a binary-portable manner and which needs to be compiled for a specific system platform should be avoided. Such software cannot easily be provisioned for a workflow to use it on different platforms.

In the present work, we extend the concept of portability to resources (i.e. data) as well. We consider a resource to be portable when it can be fully copied from one environment to another. If a text corpus can be queried using a web interface which only returns the top 100 results, the corpus data is not portable. If the service, however, offers the ability to download a dump of the complete corpus, it would be considered portable.

Some data may be considered not to be portable, because of its enormous size. E.g. the Google Web 1T 5-gram Corpus [32] has a size of several gigabytes even in compressed form, in which the data cannot be readily accessed. We accept that workflows relying on such large resources are not portable, controllable, and reproducible in the way we intend them to be – at least until hardware and network capacities have evolved to the point that the size can be handled.

Legal aspects

A license that limits or prohibits redistribution is an impediment to reproducibility, because it limits the portability. One may be technically able to provide a software or resource in a portable way, but may not be allowed to do so (cf. Figure 4.2).

```
The licensee has no right to give or sell the system to third
parties without written permission from the licenser.
```

Figure 4.2: Exemplary license clause limiting redistribution

Consider a tool with a license permitting the use in a research context, but prohibiting the downloading of the tool from any other source than the author's homepage. It would be possible to implement a portable analysis component wrapping this tool and distributing the tool along with the component to save the users from having to manually download and install the tool themselves. Due to the restrictive license, however, this is not allowed. While it is absolutely understandable that a researcher may not want others to monetarize her work, not even allowing her to redistribute the work free of charge appears to be an unnecessary impediment to research. Therefore, to ensure reproducibility, tools and data should be provided under a license which permits interested parties to repackage them and redistribute them without restriction.

```
The rights granted hereunder may be terminated at any time at
the discretion of the authors and owners. They are terminated
by the time of the eventual extinction of the FooBar Group.
```

Figure 4.3: Exemplary license termination clause

Likewise, uncontrollable license termination clauses are detrimental to reproducibility. Even if a workflow may make use of a tool at the time of writing, decay is already built in as the license may be terminated at any time, and possibly even at a determined time. Once published, the license must not be revocable, except possibly in the case where the licensee undertakes an aggressive action against the licensor. E.g. the Apache License 2.0 contains a patent retaliation clause terminating the license to an individual or group instituting a patent ligitation process.

However, portability can mitigate such problems, because users can maintain private personal or institutional archives, providing at least some in-house reproducibility.

4.1.1.2 Repositories

A repository is a central place used to share analysis components and resources used by an analysis workflow, their dependencies, and possibly the workflow itself. There is a defined protocol by which these artifacts are published and how they can be accessed. A provisioning mechanism which locates artifacts required by a particular experimental setup and installs them on the machine on which the analysis process is intended to be run can rely on this protocol. A repository can be a simple static website that hosts the artifacts, or it can be a specialized repository server software, which may offer additional features like searching over artifact metadata. Instead of visiting such a repository, manually downloading and installing artifacts, users today expect that a repository integrates with the tool they are using. For example, the add-on management dialog of the Firefox[2] browser directly accesses the Firefox plug-in repository and offers the user to install a plug-in by simply clicking on it. This entails that a repository must be programmatically accessible with reasonable ease, must provide metadata about the components it hosts, and may provide metadata about itself.

Repositories help to ensure availability. To further improve availability, it helps when repositories are organized as a federation, providing redundancy at least for commonly used artifacts. Figure 4.4 illustrates how such a federation can be structured. Such a system of federated repositories can be found in the ecosystem and the community that developed around Maven

[2] http://www.mozilla.org/firefox (Last accessed: 2013-11-12)

[180]. In the illustrated scenario, there are three main actors: the *Foo Group*, *Team Bar*, and the *community*.

The *central repository* is one of the major hubs of the repository federation. A large user community publishes their own artifacts there. This repository is itself backed by a load-balancing system and a set of mirrors to ensure availability and reliability. It is operated by a group of people dedicated to its long-term existence, e.g. a company building a part of its business model on the repository, or a well-respected non-profit organization.

The Foo Group operates two repositories, an internal one and a public one. The *private repository* serves to exchange proprietary artifacts within the group. Additionally, it serves as a caching proxy for all external artifacts used by the Foo Group. Any artifact ever used by the Foo Group remains cached there. This protects the group against connectivity failures or external repositories going out of business. For example, the *Project X* repository, hosted by a small group of people with limited time and funding, may not be available for an extended period. The *public repository* operated by the Foo Group hosts artifacts created by the group which they want to share with early adopters or cooperation partners (e.g. Team Bar), but which, e.g. for reasons of quality or licensing, are not distributed via the central repository. Additionally, it hosts any dependencies of these artifacts, which are not available from the central repository. The Foo Group publishes their high-quality artifacts to the central repository, to ensure optimal access and availability.

Team Bar operates a single repository. There, they provide all their artifacts publicly. They do not publish to the central repository, possibly because they want to retain the ability to withdraw artifacts at some point – withdrawing artifacts from the central repository is only done in cases of legal issues or similarly serious issues. Their repository also provides all dependencies of their artifacts which are not available via the central repository. Team Bar cooperates with the Foo Group and uses the Foo Group public repository to obtain artifacts from their partner. Team Bar also obtains artifacts from the central repository. While the responsibility to keep copies of artifacts is taken over by the private repository of the Foo Group, caching all artifacts from external sources, in Team Bar, every team member is individually responsible to maintain copies of any important artifacts.

This kind of system of interacting repositories and caches allows each stakeholder to maintain the desired level of control over own artifacts or those of others. If one trusts a remote repository, it can be used directly. Otherwise, copies of the artifacts can easily, even automatically, be made and kept in alternative repositories. The more interesting a particular artifact is to the community at large, the more repositories are bound to contain a copy of it.

4.1.1.3 Artifacts

The unit by which software or resources are distributed via a repository is the *artifact*. An artifact carries an identifier which can be used to identify the artifact across repositories. By convention, if an artifact has the same identifier in different repositories, it must be the same artifact. We borrow the term *artifact* from the terminology of Maven repositories. In other contexts, the corresponding term might be different, e.g. *digital object*.

In the context of this work, we are interested mainly in three types of artifacts:

- **Software** – primarily analysis components that have been integrated to work with a natural language processing framework, but also any programming libraries or possibly executables required by these components to run, including the processing framework itself.
- **Resources** – data that can be used by the analysis components, such as word lists (gazetteers), trained statistical models, etc.
- **Primary data** – data to which the analysis components are applied.

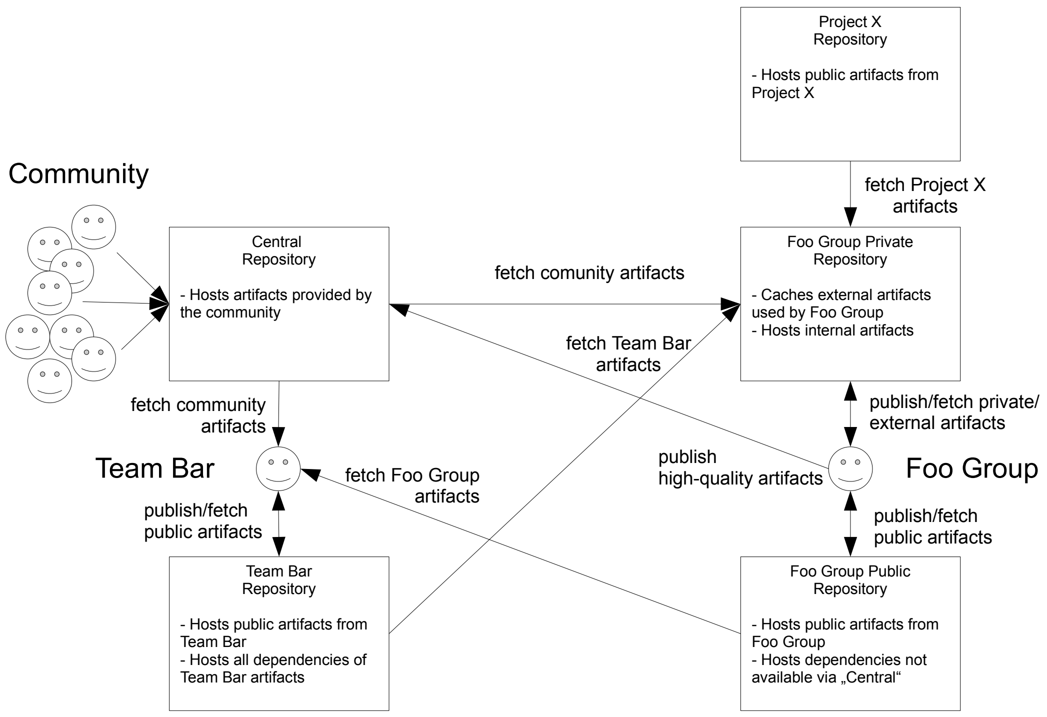

Project X Repository
- Hosts public artifacts from Project X
Community
fetch Project X artifacts
Central Repository
- Hosts artifacts provided by the community
fetch comunity artifacts
Foo Group Private Repository
- Caches external artifacts used by Foo Group
- Hosts internal artifacts
fetch Team Bar artifacts
fetch community artifacts
publish/fetch private/ external artifacts
Team Bar
fetch Foo Group artifacts
publish high-quality artifacts
Foo Group
publish/fetch public artifacts
publish/fetch public artifacts
Team Bar Repository
- Hosts public artifacts from Team Bar
- Hosts all dependencies of Team Bar artifacts
Foo Group Public Repository
- Hosts public artifacts from Foo Group
- Hosts dependencies not available via „Central“

Figure 4.4: Federation of repositories

Depending on what kind of artifact is meant to be distributed via a repository, different aspects need to be considered. These are discussed in more detail below.

Software

Analysis components are active software components to process data. When software is compiled from its source code to a binary form, the result is usually a very efficiently encoded, small output. Binaries tend to grow large only because of embedded resources, usually images, but in the case of analysis components usually because of bundled language resources. If resources required by an analysis component, e.g. the model file for a parser, are packaged as separate artifacts, the artifact containing the parser component itself can remain quite small (cf. Section 3.1). When an analysis component is required for a workflow, it is efficient and quick to fully download it from a repository and to store it on the local file system for further use.

Resources

Resources come in a variety of forms. They range in complexity from simple word list files, used as stopword lists or gazetteers, over model files for parsers or taggers and file-based knowledge sources like WordNet [157], to full-fledged SQL databases as used by the JWPL Wikipedia API [235]. They range in size from dictionary files[3] to huge resources like the Google Web 1T 5-gram Corpus[4] [32].

Not all language resources may be easily distributable as artifacts via a repository. As already mentioned in Section 3.1, additional steps may need to be taken after acquiring a resource, in order to use it. E.g. a database dump needs to be restored into a database.

The enormous size of some resources, e.g. of the Google Web 1T 5-gram Corpus may also cause technical problems, depending on the type of repository server being used, let alone the disk space available in the environment to which the resource is provisioned. However,

[3] E.g the Snowball stemmer [181] stopword list for English: approx. 4kb (`http://snowball.tartarus.org/algorithms/english/stop.txt` - Last accessed: 2013-12-02)

[4] Google Web 1T 5-gram Corpus: approx. 24 GB (compressed using gzip)

with the further evolution of technology, the size of resources that can easily be handled will inevitably increase. Not long ago, a 3.5" floppy disk with a capacity of 1.44 MB was considered top-of-the-line. On the other hand, there will always be data too large to be handled by many researchers. Still, those willing to invest the money and effort to work with this kind of data should be able to access and copy it in full, in order to maintain control over the reproducibility of their workflows. Although, the acquisition of such large data may not be as convenient, requiring manual intervention instead of automatic distribution. That notwithstanding, even such large resources should be registered in repositories under a unique identifier along with, at least, human-readable instructions on how to obtain them.

Primary data

Primary data is in many respects similar to other resources. While resources are consulted by analysis components during the analysis process, primary data is the data being subject to analysis. In order to reproduce results, this data needs to be preserved in the same way as everything else.

From a technical aspect, primary data is often easier to handle than resources, because it is structured in a simpler and more uniform way than resources, e.g. as text files encoded in one of the popular text document formats, as XML, or even as plain text.

From a legal point of view, however, primary data in natural language processing is a considerable problem. Stodden [205] states "*Raw data aren't copyrightable, and thus it's meaningless to apply a copyright rescinding license to them.*" However, unlike maybe weather sensor measurements, texts or other data produced by humans, are not such a kind of *raw data* and are subject to legal restrictions. With a few exceptions, such as Wikipedia explicitly licensing all content under a Creative Commons license, most content available on the web does either not carry a license at all, or it is some form of proprietary content. Recently, a law [99] was passed in Germany, allowing content providers to charge others, e.g. search engine providers or bloggers, for using *snippets*, short excerpts of a web page, along with search results, unless these excerpts are *sufficiently short*. However, the law does not define the actual permitted size, stating only *... einzelne Wörter oder kleinste Textausschnitte...* (individual words or smallest passages). Such legislature can basically remove the possibility of doing reproducible research on real world language samples crawled from the web, or at least it leaves researchers with a feeling of uncertainty how to do proper, reproducible research within such a legal framework. Some parties propose that anything smaller than 300 characters should be reasonably small by the standards of this law, but this has not yet been decided in court. Even then, certain analysis cannot be made on excerpts of this size. Single sentences can easily be longer than 300 characters – these may not even be parsed properly. Other analysis, like coreference resolution, requires knowledge of large passages of a document and cannot be performed at all on short snippets.

Despite such problems, bodies of primary data should be registered in repositories and there should be a way for a processing framework to automatically acquire them in full and access them. Possibly, primary data artifacts may not be accessible by the public and the user may need to be authenticated and authorized to access them. For proper public reproducibility, however, more corpora should be made available under licenses which explicitly permit public access and redistribution. Packaging primary data as portable artifacts is at least convenient within the privacy of a research group, to improve the reproducibility within the group. It may also be acceptable to the licenser of a resource that such artifacts are shared with external partners, who also possess the necessary authorization, e.g. who have purchased the license for a particular data set. However, an explicit permission should be requested from the licenser.

4.1.1.4 Discovery

To play a role in reproducible workflows, an artifact has to be uniquely identifiable. It is useful to know that a workflow uses a part-of-speech tagger, but to make the workflow reproducible, it needs to be known which part-of-speech tagger in which particular version should be used and with which particular version of a tagging model it should be configured.

Discovery is the process of locating a particular artifact based on its properties, in particular coordinates such as identifier and version (cf. Section 3.1) but possibly also other properties, such as the kind of data consumed or produced by an analysis component, the language of a data set, etc. The artifact metadata should be easily accessible via a repository.

To discover components, it should not be necessary to extract archives, analyze source code, or compiled binaries to extract the metadata. Thus, metadata should be easily readable from a file (e.g. an XML descriptor). Alternatively, a special repository software could be used which facilitates discovery by analyzing metadata contained inside the artifacts and which permits querying over this metadata.

An artifact does not necessarily correspond to exactly one component or resource. The processing frameworks we examine later allow an installable artifact – or *plug-in* – to contain more than one component. This can cause confusion because a user interested in a particular component first has to locate the artifact containing that component. Also, there could be differences between the metadata (e.g. the version) of the installable artifact and that of the components contained within.

4.1.1.5 Workflow description

The workflow description provides a processing framework with information which data to process, which analysis components should be used to process it, how these should be configured, and in which order the components should be applied. A workflow description should also be a succinct way of communicating the details of a workflow between researchers. Optimally, it should be human readable, or at least it should be displayed conveniently when loaded into annotation tools provided along with the processing framework.

Reproducibility can only be expected from a workflow description, if it exactly specifies which software and resources should be used to run it. A convenient way to do this is to include the unique identifiers and the versions of the artifacts containing these.

4.1.2 State of the art

In this section, different approaches to achieving reproducibility are presented before examining if and how these are supported by current processing frameworks.

4.1.2.1 Approaches to reproducibility

In this section, we examine different ways of ensuring reproducibility through portability, as well as of creating reproducibility without any portability at all.

Virtual machine

One approach to preserving the execution environment of a workflow is to capture the full state of the execution environment in a virtual hardware, including a bootable operating system (cf. [55]). This is an effective approach, but it is also a large waste of resources as much of the preserved environment is actually not involved in the execution of the workflow. Another drawback of the approach is that much of the environment is only preserved in binary form, e.g. software libraries used by the components involved in the workflow. While it should be possible to reproduce the experimental results with relative ease, inspecting the setup in detail or searching for bugs may be difficult without access to the source code. This approach is mainly attractive for experiments that have either been compiled to native binary code and rely on system-level libraries, or that rely on a very specific execution platform which is not easy to recreate.

A variant of this approach is the use of a virtual machine that runs programs which have been compiled to portable byte-code. Instead of virtualizing a full hardware platform, these byte-code virtual machines implement a completely hardware-independent processing model. An example of such a byte-code VM is provided by the Java Virtual Machine (JVM) specification [139]. Programs written for these virtual machines can run on any system platform for which a corresponding runtime environment has been implemented, as illustrated by the slogan "*write once, run anywhere*" that was used to promote the Java platform in its early days.

Capture and recreate environment

An alternative to capturing the full execution environment is capturing only that part of the environment which is actually used by the workflow. Guo and Engler [106] provide a tool for Linux systems called *CDE* which is used to intercept all kinds of system calls during the execution of a workflow. In that way, it captures all files, libraries, etc. which are used directly by the workflow or indirectly by spawned processes. Later, CDE can be used to run the workflow again, this time redirecting all invocation of the system calls mentioned above to use the previously captured files and libraries instead of accessing the actual file system or system libraries. This approach allows building a lightweight, self-contained version of a workflow or basically any Linux program. CDE appears to be a convenient alternative to building statically linked binaries. A statically linked binary does not call out to other, dynamically linked libraries. Instead, all functionality required by the program has been incorporated directly into the binary.

Web services

By building a workflow on web services, reproducibility can be achieved without requiring portability. Web services have the benefit of being immediately usable. No local installation of software is required. Consequently, the software running behind the web service does not need to be portable. By means of web services, software running on arbitrary system platforms can be made interoperable.

Furthermore, web services allow addressing certain legal issues. For example, services can be provided only to authorized or paying users. Access to data can be limited, e.g. to avoid the creation of exhaustive copies of databases. This way, it is e.g. possible to provide users with at least some access to data sets, which would otherwise not be accessible at all or only accessible for a fee. For example, limited access to the British National Corpus [28] is provided via a web interface[5] to anybody, while the full corpus can only be obtained by buying a corpus license. Restrictions regarding the redistribution of analysis components or resources can also be avoided by hiding them behind a web service. In this way, neither the software, nor the resources need to be redistributed. It is questionable, though, if such a web service may be used to process non-redistributable data, because the data would need be handed over to a third party which is hosting the web service.

However, the use of web services in a workflow does not only solve problems, it introduces new problems as well: the problem of reliability and the problem of versioning. Zhao et al. [237] conducted a survey of workflows shared via the social research platform *myExperiment* [103] and found that in about half of the cases, failing workflows where caused by changes in

[5] `http://corpus.byu.edu/bnc/` (Last visited: 2013-07-04)

third-party resources, such as web services. These cases were further analyzed and found to be largely due to unavailability or inaccessibility, but also to a minor degree due to upgrades.

Reliably hosting a web service over an extended period, in particular a popular service, can incur significant cost to the service provider. This cost is easily multiplied, if several versions of the service need to be offered in parallel to ensure the reproducibility of results. For this reason, we argue that portable artifacts and portable workflows provide a more sustainable foundation to reproducible research than web services. The cost of hosting a repository to distribute portable software and data should scale much better than the cost of providing each of these pieces of data or software as services.

4.1.2.2 Support for reproducibility in processing frameworks

We now examine which provisions for reproducibility are provided in the processing frameworks.

GATE

GATE allows exporting workflows as a *GATE Application* (GAPP) file. The file contains references to data storage locations, to analysis components and their configuration. In order to use the file, a GATE installation is required which provides the components and the data needs to be available in the correct locations. GATE tries to make the GAPP file more portable by storing references to files as relative paths, e.g. relative to the GATE home directory or to the directory for custom GATE plug-ins. Since references to analysis components are not versioned, it is impossible to know what version of GATE, or what combination of additional plug-ins need to be installed to reproduce analysis results.

Alternatively, GATE allows exporting an application as a package for *GATECloud.net* [209]. This package includes, in addition to the GAPP file, all GATE plug-ins used by the application, including their resources, and the corpus data.

Analysis components and other extensions to GATE are distributed as so-called *CREOLE* (*Collection of Reusable Objects for Language Engineering*) plug-ins. A CREOLE can contain one or more analysis components (called *processing resource*), language resources (*per* GATE *definition this includes corpora*), or user-interface extensions (called *visual resources*).[6] *GATE* ships with a considerable number of CREOLEs. More plug-ins can be installed from remote repositories, so-called *update sites*, after a local directory has been defined into which these plug-ins are to be installed. GATE ships with a preconfigured list of update sites and more sites can be added by the user.

Each GATE update site is described by an XML file hosted on the site (Listing 4.1). It provides a list of the CREOLE artifacts hosted at the site. Further information is then obtained by accessing the CREOLE descriptor (`creole.xml`) file within the individual artifact folders.[7] CREOLE artifacts are versioned at the level of this descriptor file. As each of the folders can only contain a single CREOLE descriptor file, it appears that GATE repositories do not support hosting multiple versions of the same artifact.

CREOLE plug-ins are described by an XML descriptor (listing 4.2). This file contains information, such as the plug-in ID, version, description, minimum GATE version required to use the CREOLE, etc. The descriptor file can cover multiple CREOLE plug-ins, aggregated into a *CREOLE directory*. Versioning and the assignment of IDs does not happen at the level of CREOLE plug-ins, but at the level of the directory. This directory-level metadata is not mandatory and

[6] For consistency with the rest of this document, we will use the term *CREOLE artifact* instead of *plug-in*, *analysis components* for processing resources, and *resources* for language resources.

[7] In GATE, artifacts are not single ZIP files, but folders.

Listing 4.1: GATE Update Site at http://creole.semanticsoftware.info/gate-update-site.xml (Last accessed: 2013-07-09)

```
<UpdateSite>
  <CreolePlugin url="http://creole.semanticsoftware.info/MuNPEx/" />
  <CreolePlugin url="http://creole.semanticsoftware.info/OpenMutationMiner/" />
  <CreolePlugin url="http://creole.semanticsoftware.info/OpenTrace/" />
  <CreolePlugin url="http://creole.semanticsoftware.info/OrganismTagger/" />
  <CreolePlugin url="http://creole.semanticsoftware.info/OwlExporter/" />
</UpdateSite>
```

Listing 4.2: GATE CREOLE XML for the *Stanford Parser* component (abbreviated)

```
<CREOLE-DIRECTORY>
  <CREOLE>
    <RESOURCE>
      <NAME>StanfordParser</NAME>
      <CLASS>gate.stanford.Parser</CLASS>
      <JAR>gate-stanford.jar</JAR>
      <JAR>lib/stanford-parser.jar</JAR>
      ... snip ...
    </RESOURCE>
  </CREOLE>
</CREOLE-DIRECTORY>
```

can mostly be found on update sites. CREOLE plug-ins shipping with GATE itself do not necessarily carry such information. For this reason, the CREOLE descriptor for the Stanford parser component (Listing 4.2) which is bundled with GATE, does not carry such information.

Instead of fully specifying the metadata in the descriptor, the metadata can be provided in the form of Java Annotations in the component source code (Listing 4.3). GATE augments the descriptor with the information from these annotations while loading the plug-in. A tool to augment the CREOLE XML descriptor during build time, to facilitate extracting such information when an artifact is placed into a repository, is not available.

In summary, even though GATE supports exporting workflows to share them with other researchers and despite its support for component repositories, the metadata is not sufficient to reconstruct the workflow environment. In particular, workflows do not bear versioned references to the used components. Components only carry versions when they are hosted in a repository. Repositories do not have the capability to host multiple versions of the same component, thus allowing users to access older versions of a component, which may have produced different results. Exports in the GATECloud.net format contain all plug-ins, but not the actual processing framework libraries. This format is sufficient to transfer a workflow to the GATECloud.net service, but it cannot even be easily imported into a locally installed GATE on another workstation to repeat an experiment.

Tesla

Tesla allows exporting an experimental workflow and handing it over to another researcher who can then repeat the experiment, given that the components and data are available. In a research group, a Tesla server can be set up on which all experiments and their results are kept and which can serve as an exchange hub between researchers. The workflow descriptor contains versioned references to all components. Additionally, hashes of the documents being processed are stored in the workflow descriptor. Consider a workflow operating on data that may not be distributed freely due to legal restrictions. Every user of the data may have to sign a separate license agreement and separately obtain the data from its original source. The hashes

Listing 4.3: GATE CREOLE Java annotations on *Stanford Parser* component (abbreviated)

```
@CreoleResource(
  name = "StanfordParser",
  comment = "Stanford parser wrapper",
  helpURL = "http://gate.ac.uk/userguide/sec:parsers:stanford")
public class Parser extends AbstractLanguageAnalyser
```

Listing 4.4: Tesla Java annotations on *Stanford Parser* component (abbreviated)

```
@Component(
  threadMode = ThreadMode.NOT_SUPPORTED,
  author = @Author(
    author = "Stephan Schwiebert",
    email = "sschwieb@spinfo.uni-koeln.de",
    web = "http://www.spinfo.phil-fak.uni-koeln.de/sschwieb.html",
    organization = "Sprachliche Informationsverarbeitung"),
  description = @Description(
    name = "Stanford Parser",
    licence = Licence.LGPL_2,
    summary = "Work in progress",
    bigO = "linear (number of Annotations)",
    version = "1.0",
    web = "http://nlp.stanford.edu/software/lex-parser.shtml",
    reusableResults = true))
public class StanfordParser extends TeslaComponent
```

make sure that Tesla can verify if the workflow is actually operating on exactly the same data for each of these users.

Tesla, being Eclipse-based, could rely on the provisioning support provided by Eclipse. This mechanism allows registering one or more *update sites*[8], special websites which host metadata describing the plug-ins available at those sites, their versions, licenses, and so on. Plug-ins offered on these websites can be downloaded and installed comfortably via Eclipse's *Install new software* wizard. Unfortunately, Tesla does not seem to make use of Eclipse's provisioning features for its analysis components.

The analysis components are all shipped embedded within the *Tesla Server* plug-in. Similar to GATE, artifacts containing the components are folders. This is probably due to the Tesla Server actually being a separate non-Eclipse product that runs separately from the Tesla workbench on a dedicated server to offload work from the machines running the workbench. On the Tesla server, any component can only be deployed in one version at a time. Even though workflows reference a particular version of a component, they can only use the version installed on the server. Using an older version of a component, which may have produced different results, is not possible.

Tesla relies exclusively on Java Annotations to describe components (listing 4.4). There are no descriptor files for components. The component's class name doubles as its identifier, while the component version is contained in the `@Description` annotation. If a component requires further resources or dependencies, those are bundled with the components.

To summarize, even though Tesla workflow descriptions make versioned references to analysis components and even reference corpus data via hashes, Tesla lacks infrastructure to fully make use of this information for reproducible workflows. There are no repositories from which Tesla components can be obtained. The Tesla server does not support multiple versions of

[8] The Eclipse update sites are built either for the meanwhile outdated *Update Manager* or its successor *Equinox p2*. Even though GATE also calls its repositories *update sites*, they are completely different from Eclipse's update sites.

the same component. While there may be sufficient information to recreate large parts of the workflow environment from the workflow description, it is by no means easy to do so.

UIMA

UIMA supports two kinds of components, *primitive components* and *aggregate components*. Primitive components are just regular analysis components, while aggregate components represent workflows, forwarding the analysis tasks to delegate components, which in turn can either be primitive or aggregate components. Component descriptors can be serialized XML files and can be shared among researchers in this way. Component descriptors, and consequently workflow descriptions, can be serialized to an XML file. Instead of embedding all the component descriptors in the same XML file, delegate component descriptors can be imported either from an URL or from the Java classpath. Such imports, however, cannot refer to a particular version of a component. When imports are not used, at least the version information of each of the delegate components can be embedded into the workflow description.

UIMA allows packaging analysis components as PEARs (*Processing Engine ARchive*[9]). These artifacts contain one or more component descriptors, any required resources and dependencies except the UIMA framework itself. A PEAR cannot be used directly from its artifact, it needs to be installed, e.g. using the *PEAR Installer* tool, before it can be used.

Currently, there appear to be no dedicated public repositories for PEAR artifacts. The UIMA Component Repository at the Carnegie Mellon University[10] had provided PEARs between 2006 and 2010, but has gone offline since 2010 according to the Internet Archive.[11] Since 2011, the Apache UIMA project distributes PEAR artifacts for some analysis components, for example the UIMA Sandbox Dictionary Annotator[12], via the Maven Central Repository along with the plain Java libraries containing the respective component implementations. Other UIMA component collections, such as DKPro Core (Section 5.2) or ClearTK [172], also distribute components as plain Java libraries. Contrary to the PEARs, these plain libraries do not need to be installed.

A PEAR only exports a single component (`SUBMITTED_COMPONENT`), all other components that may be contained in the PEAR are considered private to it. For the plain Java library version, there is no such restriction. This difference in granularity between artifacts and the components contained within, makes it more difficult for the user to determine which artifact is required in order to use a particular component. For example, the component *BreakIteratorSegmenter* in DKPro Core is contained in the *tokit* artifact, along with several other components related to tokenization. GATE allows the user to browse the components contained within a CREOLE artifact in its GUI. Users of UIMA, relying on components packaged as Java libraries, can locate the artifact containing a particular component using the classname-based search. They currently have no way, however, to comfortably browse the components within a repository.

The lack of PEAR repositories that could be programmatically accessed by the *PEAR Installer*, the need to manually locate and download PEARs, and the need to invoke the *PEAR Installer* for every single PEAR that should be installed, currently makes the use of PEARs quite complicated.

The PEAR descriptor (Listing 4.5) provides an ID for the artifact, a name, information on the installation process and how to access libraries and resources required by the analysis components within the PEAR. Most notably, there is no version information. The `SUBMITTED_COMPONENT` section refers to the analysis component descriptor of the top-level component provided by the PEAR.

[9] Getting Started: Working with PEARs: http://uima.apache.org/doc-uima-pears.html (Last accessed: 2013-03-08)
[10] UIMA Component Repository at CMU: http://uima.lti.cs.cmu.edu (Last seen offline: 2013-11-12)
[11] Last snapshot: http://web.archive.org/web/20100702180339/http://uima.lti.cs.cmu.edu/ (Last accessed: 2013-07-10)
[12] Dictionary Annotator: http://uima.apache.org/sandbox.html#dict.annotator (Last accessed: 2013-07-10)

Listing 4.5: UIMA Dictionary Annotator v. 2.3.1 PEAR descriptor (abbreviated)

```
<COMPONENT_INSTALLATION_DESCRIPTOR>
  <OS><NAME>Windows 7</NAME></OS>
  <TOOLKITS><JDK_VERSION>1.6.0</JDK_VERSION></TOOLKITS>
  <UIMA_FRAMEWORK></UIMA_FRAMEWORK>
  <SUBMITTED_COMPONENT>
    <ID>DictionaryAnnotator</ID>
    <NAME></NAME>
    <DESC>$main_root/desc/DictionaryAnnotator.xml</DESC>
    <DEPLOYMENT>standard</DEPLOYMENT>
  </SUBMITTED_COMPONENT>
  <INSTALLATION>
    <PROCESS>
      <ACTION>set_env_variable</ACTION>
      <PARAMETERS>
        <VAR_NAME>classpath</VAR_NAME>
        <VAR_VALUE>$main_root/lib/uima-an-dictionary.jar;... snip ...</VAR_VALUE>
        <COMMENTS>component classpath setting</COMMENTS>
      </PARAMETERS>
    </PROCESS>
    ... snip ...
  </INSTALLATION>
</COMPONENT_INSTALLATION_DESCRIPTOR>
```

Table 4.1: Comparison of artifact metadata

	GATE CREOLE XML	Tesla	UIMA PEAR
Root	CREOLE-DIRECTORY		SUBMITTED_COMPONENT
ID	@ID	*folder structure*	ID
Version	@VERSION	– no equivalent –	– no equivalent –
Name	(uses @ID)	– no equivalent –	NAME
Description	@Description	– no equivalent –	DESC
Additional info. URL	@HELPURL	– no equivalent –	– no equivalent –
Libraries	(only per component)	(only per component)	install.xml `set_env_variable`

Individual analysis components are described using an analysis engine descriptor (Listing 4.6). Components do not have an identifier, only a name. For primitive components, the name of the class implementing the component can be used as an identifier. For aggregate analysis components (workflows), there is no such substitute.

In summary, UIMA provides support for assembling workflow descriptions and sharing these with others. However, these descriptions are not sufficient to recreate the original workflow environment. From workflow descriptions, components cannot be addressed by versioned logical identifiers, even if the components are distributed as PEAR artifacts as well as plain Java libraries via repositories.

Summary

If a workflow description provides sufficient information to allow the recreation of the workflow execution environment, capturing the original execution environment, preserving it as a virtual machine, or resorting to web services, is not necessary to create a reproducible workflow. However, due to insufficient metadata being associated with workflows and analysis components (Table 4.2), and due to the lack of sufficient integration with component repositories (cf. Table 4.1), such portable workflows are currently not supported by the processing frameworks. In particular, the lack of identifiers and versions for components and artifacts containing these components, the missing ability to reference these from a workflow description, and the lack of means to resolve these from repositories, prevent portable, reproducible workflows.

Listing 4.6: UIMA Dictionary Annotator v. 2.3.1 analysis engine descriptor (abbreviated)

```
<analysisEngineDescription xmlns="http://uima.apache.org/resourceSpecifier">
  <frameworkImplementation>org.apache.uima.java</frameworkImplementation>
  <primitive>true</primitive>
  <annotatorImplementationName>
    org.apache.uima.annotator.dict_annot.impl.DictionaryAnnotator
  </annotatorImplementationName>
  <analysisEngineMetaData>
    <name>DictionaryAnnotator</name>
    <description></description>
    <configurationParameters>
      <configurationParameter>
        <name>DictionaryFiles</name>
        <description>list of dictionary files</description>
        <type>String</type>
        <multiValued>true</multiValued>
        <mandatory>true</mandatory>
      </configurationParameter>
      ... snip ...
    </configurationParameters>
    <configurationParameterSettings>
      <nameValuePair>
        <name>DictionaryFiles</name>
        <value><array><string>dictionary.xml</string></array></value>
      </nameValuePair>
      ... snip ...
    </configurationParameterSettings>
    ... snip ...
    <operationalProperties>
      <modifiesCas>true</modifiesCas>
      <multipleDeploymentAllowed>true</multipleDeploymentAllowed>
      <outputsNewCASes>false</outputsNewCASes>
    </operationalProperties>
  </analysisEngineMetaData>
</analysisEngineDescription
```

4.1.3 Contribution: An approach to self-contained portable workflows

The workflow descriptions of existing processing frameworks do not sufficiently support reproducibility. They put the responsibility to maintain or recreate the execution environment to the user. We envision a form of workflow description that is minimal but contains the top-level workflow logic as well as sufficient detail to provision the workflow's execution environment, i.e. analysis components, resources, and further dependencies. Such workflow description is certainly at the border between the kinds of workflow descriptions offered by current tools and dependency management, as it is done in software development projects.

We aim primarily for reproducibility, i.e. the ability to rerun the original experiment on the original data producing the original results. The experiment should run the same on the computer of a different researcher or possibly on a compute cluster. However, replacing the data with different data should be easy, and replacing individual processing steps with different implementations should also be easy. As noted by Drummond [68], to repeat an experiment, simply rerunning it to produce the exact original results, is not sufficient. In order to corroborate the hypothesis underlying the experiment, variations should be introduced into the experimental setup (cf. Chapter 4).

A collection of analysis components, such as DKPro Core (Section 5.2), providing a uniform way to try different implementations of the same processing step (e.g. part-of-speech tagging, parsing, etc.) can provide exactly the kind of variation required to elevate an experiment from being repeatable to being replicable.

Table 4.2: Comparison of component metadata

	GATE CREOLE XML	GATE CREOLE Java	Tesla	UIMA
Descriptive				
Root	RESOURCE	@CreoleResource	@Description	varies[a]
Version	(only per installable)	(only per installable)	version	version
Name	NAME	name	name	name
Description	COMMENT	comment	summary	description
Vendor	– no equivalent –	– no equivalent –	@Component(author)	vendor
Copyright/License	– no equivalent –	– no equivalent –	license	copyright
Additional info. URL	HELPURL	helpURL	web	– no equivalent –
Technical				
Root	RESOURCE	@CreoleResource	@Component	varies[b]
– Implementation	CLASS	**– annotated class –**	**– annotated class –**	varies[c]
– Library	JAR	←	directory structure	(only per installable)
– Type	INTERFACE	interfaceName	**– annotated class –**	*descriptor type*
– Scaling	– no equivalent –	– no equivalent –	threadMode	multiDeploymentAllowed

[a] The actual XML tag varies depending on the type of component. E.g. it is `processingResourceMetaData` for readers and `analysisEngineMetaData` for analysis components.

[b] The actual XML tag varies depending on the type of component. E.g. it is `collectionReaderDescription` for readers and `analysisEngineDescription` for analysis components.

[c] The actual XML tag varies depending on the type of component. E.g. it is `implementationName` for readers and `annotatorImplementationName` for analysis components.

4.1.3.1 Workflow definition

We demonstrate the feasibility of such a kind of workflow description based on a set of existing technologies. We build an analysis workflow using the convenient workflow assembly API provided for UIMA components by the uimaFIT library. We use components from the DKPro Core collection of UIMA components (Section 5.2), which requires minimal parametrization due to the use of sensible parameter defaults and the automatic resource selection mechanism (Section 3.1). Without these, a concise workflow description would not be possible. Being based on the Java Virtual Machine, the workflow is portable across platforms.

By implementing our workflow in the Groovy language, we can make it very concise. Groovy is another language based on the Java Virtual Machine. First attempts in using Groovy in conjunction with uimaFIT for writing test cases were reported by Ogren and Bethard [171]. These were not pursued further, due to the immaturity of the integration of Groovy with Eclipse IDE at the time. Meanwhile, the IDE support for Groovy has significantly improved. However, the example we illustrate here, works even without an IDE.

Compared to Java, Groovy supports convenient syntactic shortcuts, allowing for much shorter code. In fact, by virtue of the employed techniques, the workflow description is so concise and to the point that we consider it a domain specific language (cf. Chapter 3). Most importantly, however, it provides two particular features:

- **Interpreted language** – Groovy can be used as an interpreted language, which allows combining it with the interpreter directive on Unix operating systems. If an executable file in a Unix system starts with the characters `#!` (*shebang*), any text following the shebang is taken to be the path to a language interpreter. This interpreter is invoked with the path to the original executable file as parameter. Using this approach to implement a workflow as a Groovy script, we maintain a nice, human-readable description format, while at the same time creating a script that can be invoked directly from the command line.

- **Grape** – The *Groovy Advanced Packaging Engine* (*Grape*) allows declaring Maven dependencies directly in Groovy source code – no additional configuration files (e.g. a *POM* file) are required. This allows us to declare dependencies on all the analysis components used in the workflow.

These two concepts combined make the workflow fully self-contained and executable.

4.1.3.2 Execution

The requirements to execute the workflow are few. First, a Java Runtime Environment (JRE) and Groovy must both be installed. Then, access to a repository containing the artifacts required by the workflow is needed. The analysis components, along with the processing framework, are then acquired automatically. This is different from the approaches currently followed by GATE or Tesla. Both of these tools already come with analysis components pre-installed, or require the user to manually install components needed by a workflow.

It should be noted that the low-level bootstrapping tools, the JRE and Groovy, are rather stable, backwards compatible, and have little to no effect on the outcome of the workflow, even if not the exact version is used by each researcher. The analysis components, their dependencies, and resources, which are critical to the workflow results, however, are referenced by their exact version and therefore are guaranteed to be exactly the same.

The following actions take place when the workflow is executed:

- The Unix program loader invokes the Groovy interpreter, passing to it the name of the original script file.
- The Grape system resolves and acquires all dependencies and adds them to the classpath of the script.
- The Groovy interpreter executes the script.
- The reader component loads the texts to analyze. It also sets metadata, in particular the document language.
- The analysis components process each text in turn. Based on the document language, they perform an automatic resource selection processing, themselves resolving and acquiring the resource artifacts. Because components handle this dynamically, we do not need to explicitly specify Grape dependencies on these resource artifacts.

4.1.4 Example

An example script is shown in Listing 4.7. The script accepts three command line parameters:

- The directory containing the text files to process.
- The language of the texts (alternatively, an analysis component could be used to automatically detect the language of the text).
- The name of the file to which the results are written.

The first section of the scripts contains `@Grab` annotations instructing the Grape system which dependencies to use for the script. The second section contains import statements which make classes and methods accessible by their short names. The third section, finally, describes the actual analysis workflow.

We are using a UIMA-based workflow here, employing analysis components from the DKPro Core component collection (Section 5.2). The workflow starts with a reader for text files. The files are processed using a segmenter (performing sentence splitting and tokenization) and part-of-speech tagger from the OpenNLP framework wrapped as UIMA components. The final component in the workflow writes the analysis results in a format compatible with the IMS Open Corpus Workbench [77].

It should be noted here, that the workflow benefits from the dynamic resource selection and acquisition approach presented in Section 3.1. For this reason, it is neither necessary to explicitly declare the models required by the segmenter nor by the part-of-speech tagger with `@Grab` annotations, nor is it necessary to specify any additional parameters on these components. They pick up the language set on each processed document by the reader component, dynamically select and acquire the necessary resources, and perform their analysis.

Listing 4.7: Workflow scripted in Groovy

```
#!/usr/bin/env groovy
@Grab(group='de.tudarmstadt.ukp.dkpro.core',
  module='de.tudarmstadt.ukp.dkpro.core.opennlp-asl',
  version='1.5.0')
@Grab(group='de.tudarmstadt.ukp.dkpro.core',
  module='de.tudarmstadt.ukp.dkpro.core.io.text-asl',
  version='1.5.0')
@Grab(group='de.tudarmstadt.ukp.dkpro.core',
  module='de.tudarmstadt.ukp.dkpro.core.io.imscwb-asl',
  version='1.5.0')

import static org.apache.uima.fit.pipeline.SimplePipeline.*;
import static org.apache.uima.fit.factory.CollectionReaderFactory.*;
import static org.apache.uima.fit.factory.AnalysisEngineFactory.*;
import de.tudarmstadt.ukp.dkpro.core.opennlp.*;
import de.tudarmstadt.ukp.dkpro.core.io.text.*;
import de.tudarmstadt.ukp.dkpro.core.io.imscwb.*;

// Assemble and run workflow
runPipeline(
  createReaderDescription(TextReader,
    TextReader.PARAM_SOURCE_LOCATION, args[0],     // first command line parameter
    TextReader.PARAM_LANGUAGE, args[1],            // second command line parameter
    TextReader.PARAM_PATTERNS, "[+]*.txt"),
  createEngineDescription(OpenNlpSegmenter),
  createEngineDescription(OpenNlpPosTagger),
  createEngineDescription(ImsCwbWriter,
    ImsCwbWriter.PARAM_TARGET_LOCATION, args[2])); // third command line parameter
```

4.1.5 Summary

In this section, we proposed an approach to reproducible analysis workflows, by setting up a high-level description of the workflow which contains sufficient information to provision the experiment environment and run the workflow. This approach represents an improvement over current processing frameworks, which allow building analysis workflows and sharing them, but leave the responsibility of setting up the execution environment to the user. In this context, we discussed the necessity of relying on portable software for reproducibility as opposed to web services, which are another approach to removing the responsibility of setting up the execution environment from the user.

Using an example based on the Groovy language, we have demonstrated that analysis workflows can be built in such a way that they are portable, self-contained, and even executable in the sense that they contain sufficient information to allow a generic bootstrapping layer to recreate the workflow execution environment.

Our approach puts the user in control over the workflow. Whereas it has been shown that workflows based on web services are subject to decay due to services changing or becoming unavailable, our approach relies on portable artifacts. The user can maintain private copies of all involved artifacts and does not have to rely on third-parties.

The use of portable artifacts also facilitates running an analysis on a compute cluster as all involved artifacts can be automatically deployed to the cluster. No manual installation of software on the cluster nodes is required for running an analysis workflow. The DKPro BigData [61] project facilitates porting a workflow using DKPro Core components to run on an Apache Hadoop [8] cluster.[13] It benefits from concepts we presented here and from their implementation in the DKPro Core component collection (Section 5.2).

We expect that our approach fosters and facilitates the exchange of analysis workflows among researchers. For the DKPro Core component collection (Section 5.2), we already provide several Groovy-based analysis workflows as introductory examples. As part of the CSniper corpus search and annotation tool (Section 6.1), we also provide such workflows for the conversion of corpora to the formats required by CSniper.

As future work, the assembly analysis workflows for non-expert programmers should be further facilitated. While a Groovy script, as presented above, already provides a very concise representation of the workflow, it still requires prior knowledge about the components that are available, the individual parts which need to be imported from each module, and the parameters of each analysis component. An interactive tool with a graphical user interface may offer a browsable library of components to the user (cf. [129; 183]). From this library, the user could assemble an analysis workflow. Such a tool could directly save the workflow in an executable form, e.g. as a Groovy script, so that it can be easily shared with other users.

[13] The approaches presented here have enabled the development of DKPro BigData. The credits for actually realizing it, though, go to Hans-Peter Zorn, Martin Riedel, and several other early adopters at the Ubiquitous Knowledge Processing Lab and the Language Technology group.

4.2 Dynamic workflows

In this section, we present an approach to analysis workflows that change their structure based on their parametrization, e.g. in a parameter sweeping experiment which runs the same workflow with many parameter combinations. Existing processing frameworks do not readily support this concept. Generic workflow systems tend e.g. to target web services, which entail significant overhead and are problematic with respect to reproducibility (cf. Section 4.1).

Our approach has been implemented in the DKPro Lab framework [72]. It uses an open design allowing specialization for different problem domains, tools, or processing frameworks. This facilitates the automatization of auxiliary steps that are typically performed manually and therefore are prone to errors, e.g. directing output of one step of the experiment into the next one. Unlike other approaches, we focus on the programmatic assembly of complex workflows, building on existing programming skills and development environments. To illustrate the usefulness of our approach, we describe how it has been applied in research.

Our approach addresses the following issues in our overall scenario (Figure 4.5):

9 **Workflows and components are not sufficiently debuggable and refactorable.**
Our programmatic approach to workflow assembly allows building on existing programming skills and integrated development environments (IDEs). Debugging can seamlessly go from the high-level logic, over analysis components, down to the underlying libraries. When refactoring the low-level logic, the IDE can automatically update the workflow logic.

10 **Workflows that change dynamically via parametrization are not readily supported.**
Our approach provides a lightweight and non-invasive way of integrating all steps of an experiment into an automatically executable setup, it supports parameter sweeping, and it can dynamically change the structure of the workflow depending on its parametrization.

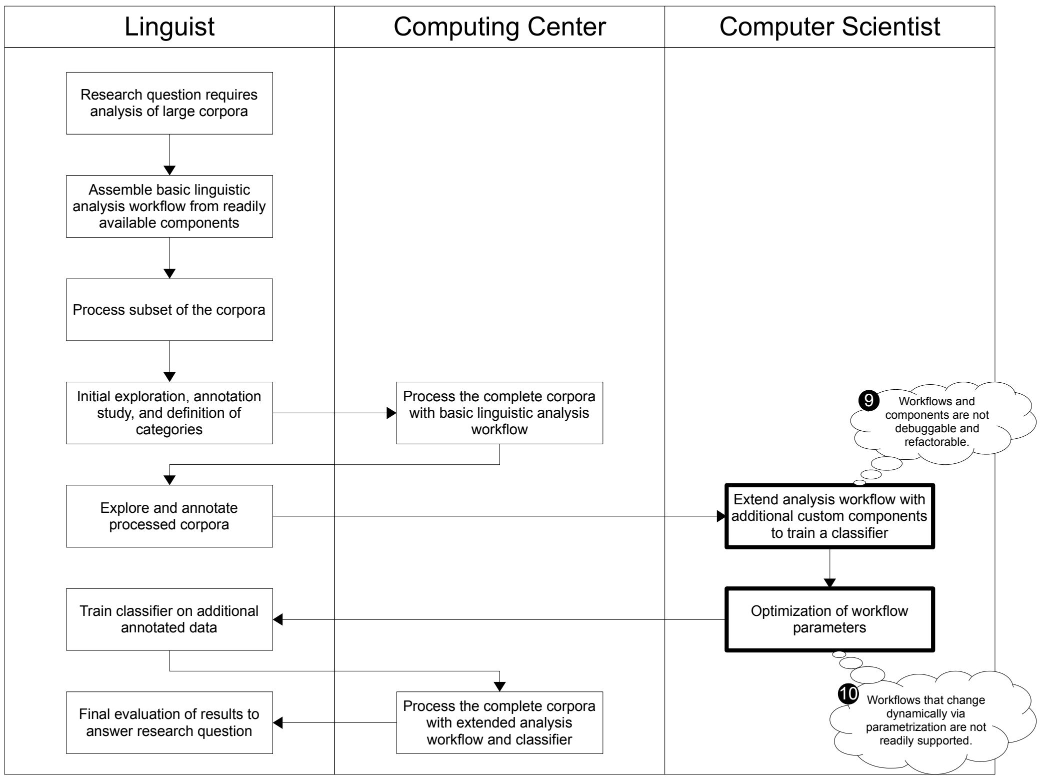

Linguist
Computing Center
Computer Scientist
Research question requires analysis of large corpora
Assemble basic linguistic analysis workflow from readily available components
Process subset of the corpora
Initial exploration, annotation study, and definition of categories
Process the complete corpora with basic linguistic analysis workflow
9 Workflows and components are not debuggable and refactorable.
Explore and annotate processed corpora
Extend analysis workflow with additional custom components to train a classifier
Train classifier on additional annotated data
Optimization of workflow parameters
10 Workflows that change dynamically via parametrization are not readily supported.
Final evaluation of results to answer research question
Process the complete corpora with extended analysis workflow and classifier

Figure 4.5: Issues addressed by our framework for dynamic parameter sweeping workflows

4.2.1 Motivation

Processing frameworks facilitate the combination of analysis components into a workflow. However, the structure of these workflows mostly resembles a simple pipeline structure – simply speaking, data is read from a source, analyzed by a sequence of components, and the results are written to a target. However, research experiments typically require more complex workflows. E.g. the output of one step of the analysis workflow becomes a resource used by an analysis component in another step, the same workflow needs to be run with several parameter combination, or data needs to be aggregated across several workflow runs. As Fei and Lu [80] note, research requires data-oriented workflows. Processing frameworks offer no immediate support for such complex workflows. Experiment workflows may even involve multiple processing frameworks, or tasks requiring tools that cannot be conveniently integrated into the processing frameworks.

Reproducibility

As a consequence, researchers split their overall analysis into manageable workflow fragments. These are executed separately and data produced by one workflow is manually fed into the next one. Such manual steps are error prone and therefore need to be eliminated from the experimental workflow whenever possible. A complex experimental setup consisting of multiple interacting analysis workflows, should nevertheless run fully automatic to provide reproducibility (see also Section 4.1).

Parameter sweeping

Parameter sweeping is the process of automatically exploring numerous plausible parameter combinations. Parameter variation plays an important role in scientific experiments [169]. An experiment can be parametrized in a way that simulates running it under different conditions, e.g. using different implementations of an algorithm or using different data sets. If the experiment yields good results under different conditions, the experiment is better suited to support the underlying hypothesis. Alternatively, parameter variation can be used to find the optimal experiment configuration.

Dynamic workflows

It is desirable to be able to reconfigure the very structure of a workflow depending on the configuration parameter values. Consider, for example, an experiment being run on two different data sets A and B. Depending on the value of the workflow parameter *dataset*, an analysis component should use either one or the other of these data sets. As these data sets may be very different, e.g. in different languages, data formats, etc., potentially very different preprocessing needs to be applied to them, resulting in different workflows. Ogasawara et al. [169] note that each parameter configuration essentially represents a new workflow.

4.2.2 State of the art

As there is considerable interest in scientific workflow systems, various systems are available from the research context. Additionally, workflow systems are used to model processes in the business context. We examine the current state of the art from the perspective of several relevant paradigms.

4.2.2.1 Control flow workflows

In control flow workflows, statements, such as conditions and loops, are the central building blocks. Split and join operations are used to mark parallelizable sections of the workflow.

The workflow mechanisms provided by GATE and UIMA support control flow workflows. In GATE, different kinds of workflows (called *applications*) can be created, ranging from simple workflows executing a series of analysis components in a fixed order, to a fully customizable workflow controlled by a Groovy script in which very complex control flows can be modeled. The most commonly used flow controller is the `FixedFlowController` which executes analysis component delegates in a fixed order. The UIMA framework provides the concept of a *flow controller* which encapsulates workflow logic. Users can provide custom flow controllers to realize arbitrarily complex workflows. If the controller determines that multiple analysis steps are independent of each other, it can issue a *parallel step*, which permits the UIMA framework to run these steps in parallel and in any order. An execution engine, such as Apache UIMA-AS [11], can use this information to optimize the use of resources, e.g. by running the analysis steps on different CPU cores.

However, in both frameworks, the workflows operate on analysis components integrated into the framework, in particular using the framework-specific data structures to exchange data between the components.

Slominski [195] examines the use of the control flow oriented *Business Process Execution Language* (*BPEL*) [126] for research workflows. Since BPEL is popular in the enterprise context, Slominski intended to reuse existing skills, resources and tools from the BPEL ecosystem. BPEL workflows involve asynchronous communication with web services and are meant to accommodate long running processes. Consequently, the workflow generally maintains a state. As part of the workflow, messages are exchanged between the involved components. As an additional benefit, BPEL workflows are not tied to any particular underlying processing framework and data structures. The scientific observation of weather patterns is provided as one scientific application where such kinds of workflows can be useful. Here, the workflow continuously observes sensory data over an extended period, tries to detect patterns as they occur, and generate alert events and reports to the researchers when required.

4.2.2.2 Data flow workflows

Fei and Lu [80] argue that research is more interested in data than in processes. Hence, data flow workflows are better suited for scientific applications, while the process-oriented control flow workflows are better suited for the modeling of business processes. In data flow workflows, the availability of input data drives the execution. The order in which the components are executed is not specified explicitly, but is implicitly defined through the data dependencies between the components. A component starts its work as soon as all required data is available. Parallelism is implicitly given when components do not depend on each others' results.

Tesla offers the user a data-oriented view on workflows. They are modeled by connecting output ports of analysis components to input ports of other components, forming directed acyclic graph of interconnected components rather than a pipeline. When a component within the workflow has produced its results, messages are asynchronously sent to components which have their data dependencies fulfilled by the new results. These components may then run in parallel, causing a better resource utilization on multi-core machines.

Due to the modular design of UIMA, in principle any kind of flow logic can be implemented. Before and after each analysis step, the current state of the analysis can be examined so that data-driven flows can be implemented. It is also possible to implement flows that take the input and output capabilities of analysis components into account. These capabilities represent mainly annotation types consumed or produced by an analysis component. The `WhiteboardFlowController` from the UIMA examples module executes an analysis step whenever its input capabilities match the combined output capabilities of any previously performed step.

4.2.2.3 Grid computing and visual programming

As Abramson et al. [3] point out, many workflow engines are built as development environments for grid computing, or as orchestration tools for web services. In addition, many of these environments adopt a visual programming paradigm. This entails that the user not only requires access to a computing grid or needs to rely on web services, but also that complex workflows with many steps are rather cumbersome to build using visual programming. While the use of web services and visual programming may allow the rapid development of comparatively simple workflows, in particular third-party web services pose a significant impediment for the reproducibility of results (cf. Section 4.1). Additionally, scaling the workflow tends to be in the focus of attention with these engines, at the expense of concepts like parameter sweeping or the dynamic assembly of workflows. E.g. UIMA-HPC [20] provides grid capabilities and a visual workflow editor, but components are wired together in a pre-defined order which cannot be changed based on parametrization. Apache UIMA-AS [11] allows scaling out UIMA analysis workflows to multiple machines in a network, but sacrifices flexibility, because the analysis components need to be pre-deployed as services on these machines. Deploying a workflow including all analysis components on demand is not possible. As a consequence, workflows cannot easily be shared with other researchers unless they have access to the same pre-deploy analysis infrastructure.

4.2.2.4 Workflow history and evolution

Several approaches focus on the evolution of the workflow itself and how to relate the results produced at different evolutionary stages to the changes made to the workflow. For example, *VisTrails* [92] supports the interactive development and evolution of analysis workflows with a focus on the visualization of results. The concept entails that all results produced within the system are recorded along with the workflow and the parameters that produced them, a feature which improves development turn around times as the results can be reused if portions of a workflow and its parametrization remain unchanged in between executions. This entails that results, once they have been produced, must not be changed again. Fei and Lu [80] call this the *single-assignment property*. Slominski [195] postulates such concepts as specific requirements for scientific workflows towards experimental flexibility, history, and provenance. In fact, instead of manipulating and evolving a workflow manually, parameter sweeping can be seen as a way to automatically explore a set of workflow variations, and thus has similar requirements with respect to tracking results and relating them to the particular parametrization by which they were obtained.

GATE is also capable of saving analysis results in a data store and loading them again later for further processing, manual analysis or examination. This function, however, is not integrated into workflows, e.g. to store intermediate results which could be inspected or reused, but rather allows saving the results of an entire workflow. Another workflow can read documents directly from a data store, but it writes the analysis results back to the same store. Hence, the single-assignment property is not given. This makes setting up a chain of workflows where each step reads data from one store and saves results to another is not possible or at least fairly complicated.

When an experiment is run in Tesla, the results of each component are persisted in a database. Should the component ever be run in the same configuration on the same data and with the same preprocessing, these results can be reused instead of being recalculated. Tesla protocols the run of every experiment including which components were used in what version, how their parameters were set, and to which data they were applied.

Workflow composition

Another important feature for a workflow system is the ability to compose workflows from other workflows. Slominski [195] subsumes this under the heading *reuse and hierarchical composition*.

UIMA supports the notion of an *aggregate analysis component*. This is an analysis component which performs no analysis by itself but delegates the analysis to a number of other analysis components. These delegates, in turn, can again be aggregates, or they can be primitive engines, that actually perform the analysis. Configuration parameters of delegates can be exposed as parameters of the aggregate. This allows the creation of complex, reusable, but still configurable aggregates.

In Tesla, embedding a workflow as a component in another workflow is not supported.

In GATE, the different kinds of workflows are realized as implementations of the `LanguageAnalyser` interface. The same interface is also the basis for analysis components that can be added as delegates to a workflow. Consequently, one workflow can be embedded into another workflow in the role of a delegate analysis component, allowing the creation of reusable, complex workflows. However, there is no support for configuring the delegates of a nested workflow, limiting the reusability of complex workflows.

4.2.2.5 Workflow descriptions

Workflows are typically represented by a descriptor, typically an XML file which specifies the characteristics of the workflow, e.g. which analysis components are used, how they are configured, and in which order they are executed (cf. Section 4.1.2.2). This is particularly convenient for a system that provides graphical tool support of for configuring components and assembling workflows. It is relatively easy to implement such tools on a declarative model of the workflow.

However, these descriptions often refer to implementation details of the underlying analysis components, e.g. class names, parameter names, etc. which are subject to change during the development and refactoring of workflows and components. There has been criticism towards descriptor-based approaches, because the refactoring support of IDEs does not update such XML descriptions when the underlying implementations change and there is no way for the IDE to check for errors at compile time (cf. [171]).

An alternative would be to focus on the programmatic creation of workflows and to use the program code itself as the workflow description. Such an approach would require the workflow engine to offer a concise programming interface, as otherwise the workflow description can easily become hard to read.

4.2.3 Contribution: Dynamic workflows for language analysis

Current workflow systems used in language analysis offer many capabilities, from scaling to massive amounts of data, over visual workflow editors, to fully customizable flow logic using scripts. However, none of them facilitates the building of parameter sweeping experiments and the dynamic assembly of workflows based on parametrization. In order to address this, we designed a lightweight workflow framework called *DKPro Lab* [72]. We neither aim to provide yet another development environment, nor a replacement for sophisticated grid workflow systems. Instead, we propose a lightweight framework which can easily be used for building dynamic analysis workflows, independent of the underlying processing framework. For this reason, we also forego grid computing, scaling out, or other parallelization at this point, although the open design of the framework should allow adding such capabilities at a later point in time.

Goal

Our main aim is to enable researchers to go beyond the capabilities of the processing frameworks currently used for language analysis. DKPro Lab is meant to introduce users to the concepts of data-flow-based workflows, allowing them to gradually convert existing Java-based ad-hoc experiment code into a more organized form resembling tasks interacting via data dependencies. The framework focuses on the programmatic creation of workflows. It provides neither a graphical user interface, nor does it use any kind of XML-based workflow description. It allows the user to add a thin high-level orchestration layer on top of existing experimental code, which can be refined step-by-step as the users gets acquainted with the framework and the concept of building complex workflows. In this way, we separate the high-level workflow logic from the lower-level implementations or particular analysis workflows, and improve readability at the high level. Similar to Slominski [195], who examines BPEL as a workflow language for scientific workflows, we focus on the programmatic creation of workflows using Java in order to reuse resources and skills. While Slominski plans on reusing books and tools from the ecosystem around BPEL, we aim to reuse integrated development environments, debuggers, and programming skills for the Java platform. Our work should serve as an easy to use, yet powerful, entry into the building of more complex workflows. After some time, with increasing familiarity with the concepts and increasing requirements, a user may turn to more sophisticated systems, e.g. those with grid computing capabilities.

Benefits

The reason why users are motivated to work with DKPro Lab in their experiments is because it significantly facilitates parameter sweeping – a capability that the processing frameworks used for simple analysis workflows do not support.

What distinguishes the approach taken by DKPro Lab from other workflow engines, is the strong focus on the programmatic creation of workflows. Compared to using static workflow descriptions, the programmatic approach allows a workflow to react very flexibly to parametrization, because the parameters drive a high-level workflow logic implemented in a general purpose programming language. As a consequence, parameters in DKPro Lab can affect anything from the components, over the flow of data, to the dynamic generation of new sub-workflows. In other workflow engines, in particular those provided by processing frameworks, parameters are passed directly to the analysis components within the workflow but cannot affect the structure of the workflow itself.

A lightweight approach

We consider DKPro Lab to be a lightweight framework. Although the terms *lightweight* and *heavyweight* are widely used in literature, there appears to be no widely accepted definition of them. An interesting approach is the one suggested by Lloyd et al. [141] who measure the *weight* of a framework which allows comparing different frameworks to each other. However, they do not define absolute ranges in which a framework would be considered lightweight or heavyweight. In general, frameworks seem to be *lightweight* when they are non-invasive, focused, unobtrusive, and rely on existing concepts as much as possible. On the other hand, *heavyweight* frameworks tend to require invasive changes when they are integrated with existing code, introduce a significant set of new concepts which need to be learned, and offer features which are not related to their core functionality. By these standards, we consider frameworks like UIMA or GATE to be heavyweight.

For DKPro Lab, the following design decisions have been made to keep the framework as lightweight as possible:

- **Modular design** – Non-essential functionalities have been moved to optional modules, thus keeping the core DKPro Lab framework as focused as possible. Such modules offer, for example, improved support for the Groovy language or support for UIMA workflows.
- **Parameters can be of any data type** – DKPro Lab supports basically any data type for parametrization. The only restriction is, that the string representation of two values needs to be the same if the two values are the same, and different otherwise. The string representation is obtained via the *toString()* method the respective Java classes.
- **Mediating layer between the high-level experiment setup from low-level logic** – DKPro Lab defines the *Task* (see below) as a specific concept which mediates between the DKPro Lab concepts, e.g. the parameters, workflow lifecycle, etc. and the low-level workflow implementation. As a consequence, no invasive changes to the low-level workflow implementation are necessary when it is integrated into a DKPro Lab workflow.
- **Using the file system as storage layer** – DKPro Lab provides a storage abstraction layer which uses the file system. As a consequence, it is easy to integrate any kind of process into DKPro Lab, which can read data from the file system and write its results there. Early versions of DKPro Lab used a database for storage and enforced access to the data using Java streams. However, this was considered too restrictive and would have required invasive changes to experiment code that was converted to using DKPro Lab.
- **Programmatic approach** – DKPro Lab workflows are Java or Groovy programs. This allows using facilities of integrated development environments (IDEs) for debugging and refactoring. E.g. the low-level logic is refactored, the IDE can automatically update the workflow logic. When debugging an experiment, the whole path from the high-level experimental setup down to the low-level logic and underlying libraries can be conveniently inspected.

Because of these design decisions, it is easy to wrap existing experiment code as a DKPro Lab workflow, e.g. integrate previously separate experiment steps into a single comprehensive setup, or to add support for parameter sweeping.

Additionally, the development of DKPro Lab was guided by concrete research needs and the feedback from users. E.g. method names in the API, error message, and logging functionalities have been improved several times to make them easier to use and more understandable.

Assumptions

For adoption, it is essential that the framework is lightweight, easy to use, and interferes as little as possible with the programming behavior of the user. Hence, the framework enforces practically nothing and assumptions are also kept to a minimum. In order to keep the framework as lightweight as possible, the following assumptions are made:

- **Same in, same out** – Given the same parameters, the workflow and any step or component within it always produce the same output. In some contexts, a dynamic workflow would be expected to react to an external event, e.g. a failing grid resource or web service. However, this is not the kind of dynamics supported by this framework. We consider a workflow to be dynamic when its structure can change depending on its parameters.

- **Run to end** – A workflow starts, runs, and terminates. It should not start and run indefinitely or wait for some external event to occur before stopping. DKPro Lab is not a large-scale workflow engine running as a dedicated service with the aim of handling multiple parallel workflows possible from different users. It neither tries to support very long running, asynchronous workflows.

DKPro Lab does not make any assumptions about the structure of a workflow as defined by the data dependencies. By using a just-in-time planning approach which determines the next processing step only when the previous step is complete, DKPro Lab allows the data dependencies to change dynamically depending on the parametrization. With static planning, the workflow system can derive an execution plan once and run it. However, this requires that the data dependencies do not change during execution. A drawback of the just-in-time approach is, however, that it is unknown if a workflow can run, e.g. if it contains any unfulfillable dependencies or loops, until it is actually run.

Building blocks

Experimental workflows built with the DKPro Lab framework consist of the following basic building blocks:

- **Tasks** – A task is a specification of something that needs to be done. The framework does not prescribe the exact form of such a specification, but assumes that it is a concise piece of code written in an embedded DSL (cf. Chapter 3) or as a brief Java program. The framework supports grouping similar tasks into task categories which may provide mini-DSLs for tasks of the particular kind. Such a category typically consists of an interface, an abstract implementation of that interface which may provide the mini-DSL to concrete sub-classes, and a *task engine* which executes the task specification. The most important category is the *batch task*, which has subtasks and performs a parameter sweep over them based on the *parameter space* with which it has been configured. Tasks can have *parameters*, and they can access results produced by other tasks or external data via *data dependencies*.

- **Parameters** – The parametrization of tasks involves *parameter dimensions* which have a name and produce a finite number of parameter values. These dimensions are used to populate a *parameter space* which represents the set of all possible parameter value combinations. Parameter space and dimensions together define the order in which the parameter value combinations are explored.

- **Data dependencies** – A data dependency declares that a task depends on a particular result from another task. When a task is executed, it runs within a context specific to this single execution. The task context is similar to a key-value store, where results can be stored and data can be read from. A data dependency declares that access to specific key is redirected to access results produced by a previously run task. This entails that a task with data dependencies can only be executed after all tasks it depends on have been run.

- **Reports** – Reports post-process the results produced by a task to produce informative charts, overviews, or other forms of presentation. The results created by a task may not be immediately suited for inspection. In the worst case, it may be binary or compressed, but typically, it is just extremely detailed. Reports extract important information from these results to provide the researcher with the interesting facts, e.g. comparative overviews over results generated from different parameter combinations.

Figure 4.6 schematically shows a hypothetical workflow illustrating how these building blocks interact.

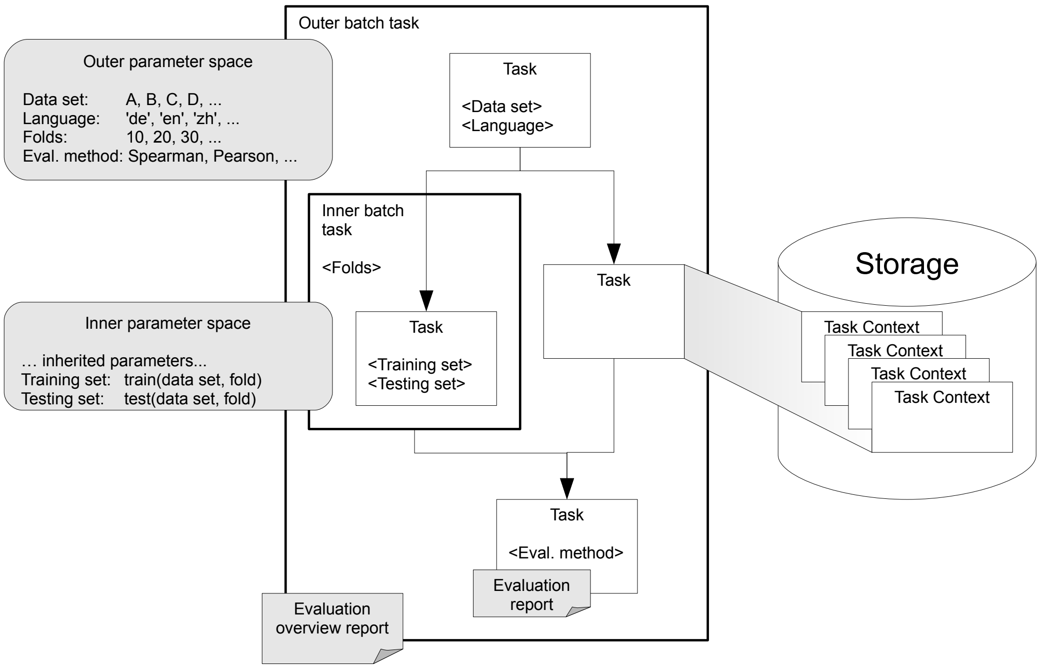


Figure 4.6: Schematic illustration of a workflow in DKPro Lab consisting of an outer and an inner batch task, each with a parameter space, reports for post processing evaluation results, and a storage layer tracking the results of each task execution during the parameter sweep

In the remainder of this section, we give a more detailed explanation of the building blocks and how they can be specialized towards certain problem domains.

4.2.3.1 Tasks

A task is the specification of what should be done. Within the DKPro Lab framework, the task serves as an adapter between the parameter space and the parametrized logic. Because a task is typically tuned to a particular parameter space and parameter spaces tend to be specific for a particular experiment, tasks are normally not reused between different workflows, unless these workflows are very similar. However, certain kinds of tasks appear repeatedly across all kinds of experiments, e.g. invocations of analysis workflows for data preprocessing, parameter sweeping, feature extraction, etc.

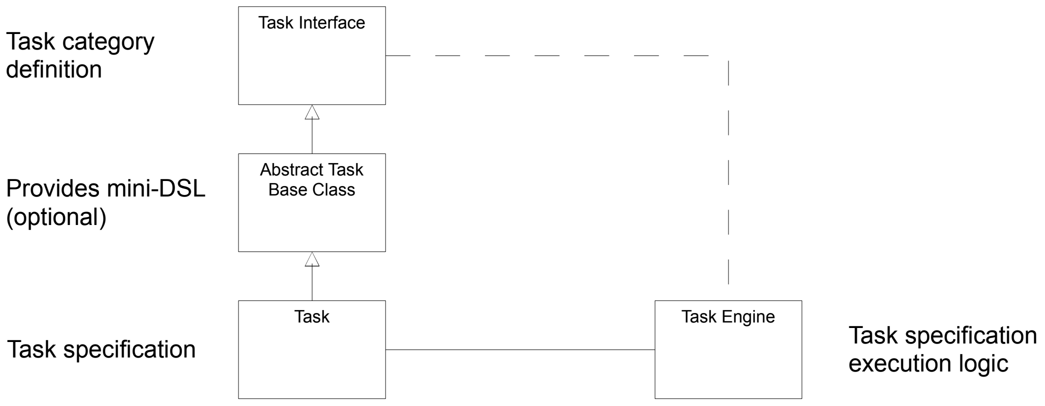


Figure 4.7: Task category: task interface, task base class, task, and task engine

Listing 4.8: Minimal DKPro Lab workflow

```
def task = new ExecutableTask() {
  void execute(TaskContext context) {
    println "Hello world!";
  }
}
Lab.instance.run(task);
```

To facilitate working with common kinds of tasks, task categories can be defined. A task category consists of an interface, an abstract base class of that interface and a task engine (see Figure 4.7). To define a task within a workflow, the user derives a task class for this particular workflow from the abstract base class. This base class may provide convenience methods that form a mini-DSL, which facilitate the task specification in the derived class. All logic commonly performed by tasks of this category can be extracted into a task engine. When the task is executed, the framework locates the associated task engine and passes the task to it. The engine retrieves all information from the task that it needs to run it according to specification. A user can register new categories of tasks (i.e. interfaces and engines) with the framework, and thus specialize it for a particular problem domain or facilitate the interaction with particular processing frameworks or tools. Currently, the framework provides immediate support for the following task categories:

1. **Executable task** – This is the simplest category of task meant to wrap any arbitrary piece of existing code. The task interface defines a single `execute()` method which needs to be implemented, similar to the `main()` method in Java programs. Listing 4.8 shows a minimal workflow using an executable task.

2. **Batch task** – This task category allows running multiple related sub-tasks. The relations between sub-tasks are modeled as data dependencies. It also allows performing parameter sweeping experiments.

3. **UIMA task** – This task category is specialized for running analysis workflows using the UIMA framework. The task interface defines methods for declaring the reader[14] component and the analysis[15] components of the UIMA workflow. The UIMA workflow descriptions are persisted as XML files and more readable HTML files. An abstract base class for a UIMA task provides a mini-DSL for a concise definition of UIMA workflows.

There can be multiple engines per task category. For example, for the UIMA task, we provide two different engines, one running the UIMA analysis workflow in a single thread, and another one using multi-threading. For development and debugging, the single-threaded engine is better suited, while the multi-threaded engine can speed up processing. In future work, we plan to provide alternative engines that allow running workflows in a cluster environment. The ultimate aim is being able to run the same workflow on a workstation or on a cluster environment just by exchanging the task engine implementations.

4.2.3.2 Data dependencies

A workflow consisting of isolated tasks is of little use. Rather, workflows are defined by the data that flows between the tasks. The way in which data is allowed to flow is defined by *data*

[14] `getCollectionReaderDescription(TaskContext)`
[15] `getAnalysisEngineDescription(TaskContext)`

dependencies, meaning that one task depends on a certain information produced by another task. Data dependencies are illustrated in Figures 4.9 (p. 96) and 4.10 (p. 98) as the connections that cross task boundaries. E.g. the *classify* step in the evaluation task (Figure 4.10) has a data dependency on the *classifier* output of the training task. Data dependencies can be used to plan the execution order of the workflow tasks. The data dependency in the given example implies that the training task has to be executed before the evaluation task.

Data dependencies between tasks

Every execution of a task takes place within an *execution context*. This is realized by creating a folder on the file system for each task execution. By convention, any results produced by the task should be stored within this folder. In many cases, it is easy for the researcher to parametrize analysis workflows or other code to write results to a certain folder. Being lightweight, the system does not enforce this, nor does it monitor system calls to intercept access to resources stored outside this folder.

Once results have been produced, they must not be changed again. Fei and Lu [80] call this the *single-assignment property*. This allows the users to inspect intermediate results produced by the different parametrizations of each task. It also allows the framework to detect if a task has already been executed with a particular parametrization. If this is the case, the results from the previous execution are reused to speed up the execution of the workflow.

In practice, however, the results of a task execution sometimes need to be updated or augmented. E.g. additional information needs to be added to a file produced by a previous task or additional files need to be placed next to files produced by a previous task. Consider a log file which is passed on from task to task with each task writing additional logging information to the file. At the end of every task execution, the log file contains a trace of all executions leading up to this point, but not any information from unrelated task executions, e.g. from executions belonging to different coordinates in the parameter space.

With some cooperation from the users, the framework can support this while maintaining the single-assignment property without invasive mechanisms. To this end, whenever a result from another execution context is accessed via a data dependency, an *access mode* needs to be declared:

1. **Read-only** – a read only access is simply redirected to the folder of the other execution context. By convention, data accessed in this way must not be changed. The framework does not enforce this.

2. **Read/write** – if write access to the data is required, the framework copies the data from the source execution context to the current context before proceeding.

3. **Add-only** – this mode can be used if the current task reads existing files from another execution context and writes new results to new files besides the already existing ones. In this case, the framework tries to avoid copying the data from the source context by creating symbolic links instead. This approach can drastically reduce the required disk space and decrease execution time. In case the underlying operating system does not support symbolic links, the framework falls back to copying.

Data dependencies on external data

A task can also depend on online resources, ranging from simple files to web services offering data via an URL. Since this data may change, the framework stores a copy of such data as part of the execution context. Subsequent runs of the workflow can be configured either to use this data or to acquire it again.

4.2.3.3 Parameters

Parameter space

The parameter space is the space created by all possible parameter combinations. Each parameter forms a dimension in this space and is represented by the values a parameter can assume. A point in the parameter space specifies a unique binding of values to parameters. Listing 4.9 illustrates the definition of three dimensions called *numbers*, *strings*, and *lists*.

In our approach, parameter values are not limited to primitive data types. Any Java object can be used as a parameter value, provided that its value can be represented as a unique string. In the current design, a dimension is always made up of discrete values, e.g. a list of numbers, strings, objects, etc. Although it is currently the case that the size of a dimension is known, the framework does not build on that assumption. We mean to maintain the option of having dimensions with an undetermined size, dynamically creating new values to explore certain areas of the parameter space in more detail, e.g. to find those parameters that optimize a certain objective function.

Listing 4.9: Discrete parameter dimension declarations

```
def dimNumbers = Dimension.create("numbers", 1, 2, 3, 4, 5, 6, 7, 8, 9, 20);
def dimStrings = Dimension.create("strings", "one", "two", "three");
def dimLists   = Dimension.create("lists", [1, 2], [3, 4], [5, 6]);
```

Parameter bundles

It is often the case that parameters are not independent of each other. Consider a data set from the IR scenario that contains documents, queries, and judgments. In practice, these are three different parameters, e.g. paths to the folders containing the respective data. If the documents come in different formats, a fourth parameter may be the component used to read the documents into the analysis workflow. A fifth parameter may be the language of the data set, which controls what analysis components or models are used during preprocessing. So for what appears to be a single parameter, e.g. the data set, in practice, we end up having multiple parameters. Of course, exploring the space of these parameters is hardly useful, e.g. testing how well a preprocessing for English works on a set of German documents. Interdependent parameters can be added as a *bundle* (Listing 4.10) to the parameter space, allowing the user to pre-define certain parameter combinations that should be used, while all others are skipped.

Listing 4.10: Bundle parameter dimension declaration

```
def dimDataSet = Dimension.createBundle("dataSet",
  [
    language:   "de",
    reader:     PdfReader,
    documents:  "/data/ir/set1/documents" as File
    queries:    "/data/ir/set1/queries" as File
    judgments:  "/data/ir/set1/judgments" as File
  ], [
    language:   "en",
    reader:     TextReader,
    documents:  "/data/ir/set2/documents" as File
    queries:    "/data/ir/set2/queries" as File
    judgments:  "/data/ir/set2/judgments" as File
]);
```

Parameter constraints

Let us assume we would like to use different stopword lists depending on the language of the data set. Of course, it would be easy to add the stopword list as an additional parameter in the data set bundle. However, if we have multiple data sets using the same language, we may prefer not to repeat the stopword list over and over again in every bundle. In this case, *constraints* are another way of controlling which parts of the parameter space should be explored (Listing 4.11). If constraints are defined in a parameter space, only those parts of the space are evaluated for which the constraints evaluate to `true`. If more than one constraint is defined, they are treated as a disjunction (their values are combined using a logical *or* operation).

Listing 4.11: Parameter constraint declaration

```
def dimStopwords = Dimension.create("stopwords",
  "/data/ir/stopwords-de.txt", "/data/ir/stopwords-en.txt");

def stopwordsConstraint = new Constraint({
  it["stopwords"].endsWith(it["language"] + ".txt")
});

def pSpace = [
  constraints: [stopWordConstraint],
  dimensions:  [dimDataSet, dimStopwords]
] as ParameterSpace;
```

Task parametrization

A task serves as an adapter between the high-level workflow modeled within the framework and the low-level logic responsible for the actual processing. Within a batch task, the task functions as an adapter between the workflow-level parameter space and the parameters of the lower-level implementation.

The task also serves as an adapter between the parameter space and the actual logic. The framework automatically injects parameter values into fields which have been marked as discriminators. Only such fields are used to determine the execution order of tasks. To have the framework inject parameter values into a field without having any effect on the execution order, they can be marked as an attribute.

Batch task

The batch task allows the user to run multiple interacting tasks. Tasks are connected to each other via data dependencies. The batch task automatically detects in which order the subtasks have to be executed. It visits each coordinate in the parameter space in turn. At each coordinate, a queue is constructed consisting of all subtasks. If all prerequisites for the top task in the queue are met, it is executed. Otherwise, it is put at the end of the queue. This approach allows the data dependencies to change at runtime depending on the parameters. Batch tasks can be nested inside each other. A nested batch task inherits the parameter space from its parent.

For example, an experiment which contains two tasks (A and B) that preprocess the data set in different ways. A third task C analyzes and evaluates each of these data sets. A parameter p controls if task C should use either the preprocessed data produced by task A or the one from task B. Thus, task C dynamically changes its data dependency on task A or B depending on this parameter (Figure 4.8).

Listing 4.12: Minimal DKPro Lab parameter sweeping workflow

```
def subtask = new ExecutableTask() {
  @Discriminator int x;
  @Discriminator int y;

  void execute(TaskContext context) {
    println "Parameter space coordinates x: ${x}  y: ${y}";
  }
}

def batchTask = [
  parameterSpace: [
    dimensions: [
      Dimension.create("x", 1, 2, 3, 4, 5),
      Dimension.create("y", 1, 2, 3, 4, 5)],
    tasks: [subtask]
  ] as ParameterSpace;
] as BatchTask;

Lab.instance.run(batchTask);
```

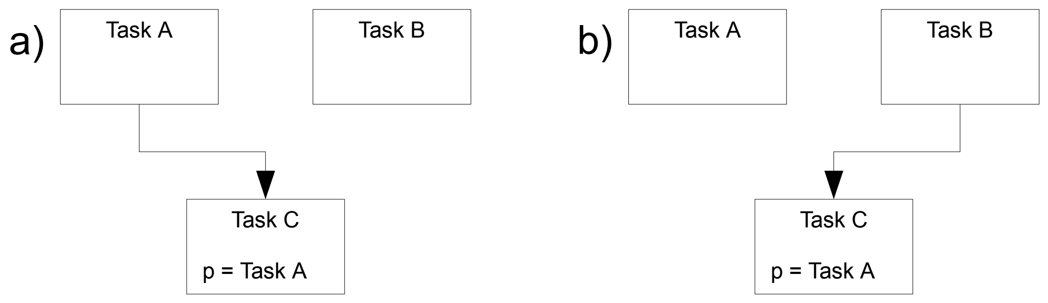


Figure 4.8: Data dependency dynamically configured via parameter *p*

4.2.3.4 Reporting

The results produced by a task may not be suited very well for human consumption, e.g. long XML files, exhaustive CSV files, etc. It is the purpose of *reports* to post-process such results and create charts, excerpts. Multiple reports can be added to a task. By convention, a task should never declare a data dependency on output produced by a report. The results produced by reports should be completely separate from the results produced by the actual task logic. Given the output of a workflow execution, it should be possible to re-run only the reports, e.g. to create a better visualization or interpretation of the data without having to re-run the full workflow again. As such, reports are meant to exist as an additional layer on top of tasks. Again, the framework does not enforce this convention.

4.2.3.5 Specialization

The DKPro Lab framework focuses on the core necessities for workflow building, but provides an open design that allows users to specialize it towards certain problem domains.

- **Reporting** – The visualization of experiment results is performed by reports that can be attached to tasks. For every problem domain, certain reports are typical, e.g. precision/recall plots for information retrieval experiments, or confusion matrices for machine learning experiments. It is convenient to build a set of reports and associated data types and reuse them across different experiments of the same kind. Eckart de Castilho and

Gurevych [72] have built various reports for information retrieval experiments. DKPro TC [65] provides reports for text classification experiments.

- **Task categories** – When certain kinds of tasks are very common in a particular problem domain, users can create specialized versions by subclassing existing task categories or even by creating completely new task category interfaces and corresponding task engines. For example, DKPro TC [65] provides convenient specialized tasks for text classification experiments: 1) a task gathering global information about the data being processed, e.g. TF/IDF counts, 2) a feature extraction task, 3) a preprocessing task, 4) a training and evaluation task.

- **Parameter dimensions** – It is possible to implement custom parameter dimensions. For example, to support cross-validation experiments, a specialized `FoldDimension` has been implemented which is wrapped around another dimension and partitions its values into n subsets of which $n-1$ can be used for training and the remaining one for testing an algorithm. The `FoldDimension` returns n different folds, each time with a different combination of the subset used for training and testing.

- **Parameter space** – We expect that future work may also require specialized parameter space implementations. Currently, each coordinate within the parameter space is visited during the processing of a parameter sweeping experiment. Obviously, this does not scale to parameter spaces with millions of parameter combinations. Alternative strategies to navigate the parameter space will be required, e.g. optimization strategies.

4.2.4 Examples

To illustrate the usefulness of dynamic workflows in more detail, this section examines two concrete experiment scenarios: 1) an *information retrieval experiment* and 2) a *machine learning experiment*.

Both scenarios originate from research undertaken at the Ubiquitous Knowledge Processing Lab. DKPro Lab was conceived while the author was working on the first scenario Eckart de Castilho and Gurevych [72]. It also serves as the basis for the DKPro Text Classification [65] project targeting the second scenario.[16]

DKPro Lab has been used in additional tasks, which we do not further explain here. Ferschke et al. [85] used it in a system for predicting quality flaws in Wikipedia. Daxenberger and Gurevych [56] used it in experiments on automatically classifying edit categories in Wikipedia revisions. Flekova and Gurevych [87] used it for a large scale age and gender author profiling study in social media. Zesch and Haase [234] used it for research on preposition and determiner correction. Zesch et al. [236] used it in a system using text similarity metrics for the textual entailment task.

4.2.4.1 Scenario 1: Information retrieval

Information retrieval (IR) deals with the task of retrieving information satisfying the information need of the user, which is expressed as a query statement . Given such a query, an IR system locates relevant pieces of information, e.g. documents, and offers them ranked by relevance to the user. An IR system performs well when those documents satisfying the information need best are ranked highest, if no relevant documents are missed, and if irrelevant documents are

[16] The credits for DKPro TC go primarily to Oliver Ferschke, Johannes Daxenberger, and Torsten Zesch. The author has continued to extend and improve DKPro Lab in the context of DKPro TC.

not retrieved. The DKPro Lab framework was used in the context of information retrieval by Eckart de Castilho and Gurevych [73]. The experimental setup itself is discussed in Eckart de Castilho and Gurevych [72].

Workflow structure

There are three tasks within the experimental workflow: 1) preprocessing and indexing the documents, 2) preprocessing the queries, and 3) searching the index and evaluating the results (Figure 4.9). Because IR systems use global information about the indexed documents, such as term frequency counts, the indexing processing needs to be complete before any search can be performed. The indexing task reads the documents from the data set, processes the data, and writes the processed data to an index. Queries are preprocessed as well, e.g. to remove stopwords, expand the query with related terms, etc. To avoid repeating this process over and over during the search and evaluation, it is modeled as a separate task which creates a set of preprocessed queries. This index and the preprocessed queries are inputs to the evaluation task, which reads the queries, retrieves relevant documents from the index, and evaluates these against the human judgments.

Each task can be easily modeled as a workflow using a processing framework. The processing frameworks support workflows which read a document and pass it through a series of analysis components. However, modeling the interaction of the tasks is not easily possible because each task needs to have processed all data before the next task can begin. The processing frameworks do not readily support workflows which consist of multiple sub-workflows that need to wait for one another.

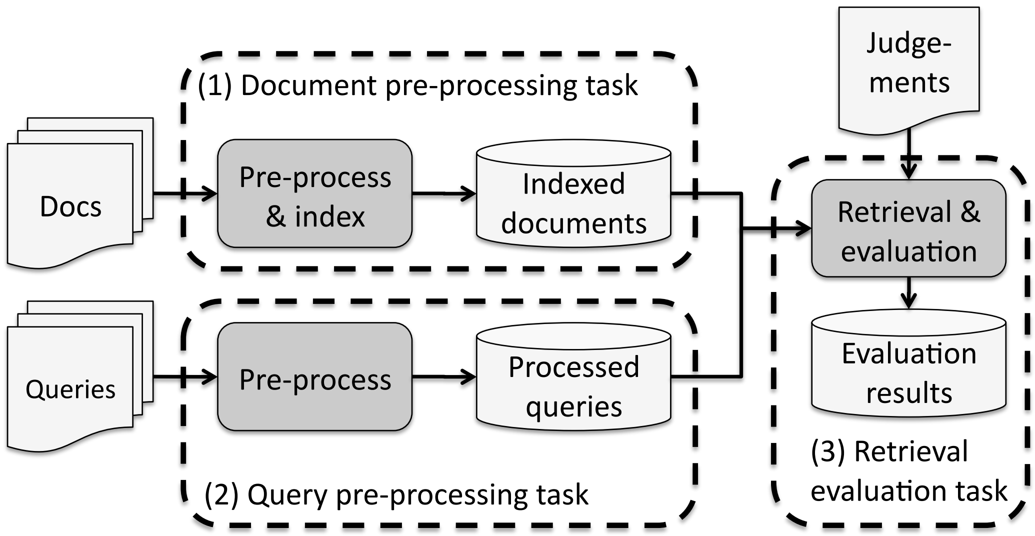


Figure 4.9: Information retrieval experiment workflow by Eckart de Castilho and Gurevych [73]

Parameters

The ranking function is at the heart of each IR system. Choosing the best parameters for this function is essential to achieve good results. However, before data even reaches the ranking function during retrieval, it is preprocessed and indexed. The indexing ensures that data can be retrieved quickly, while the preprocessing normalizes or enriches the data. For example, IR systems relying on the matching of terms, are unable to determine that a document containing the term *children* may be relevant to a query containing the term *child*. A preprocessing step called *stemming* needs to be applied to documents during indexing and queries while searching to normalize query and document terms so that they can be matched. Alternative approaches enrich queries by adding terms semantically related to those query terms provided by the user. E.g. a query for *black bird* may be extended by the term *raven* so that documents about ravens, which are black birds, can be retrieved. So, the IR experiment roughly includes the following parameters:

- **Data set** – a set of documents, queries, and human judgments indicating which documents are relevant to which queries.
- **Ranking function** – the function determining the relevance of a document to a query, possibly with additional function-specific parameters.
- **Document preprocessing** – the enhancements and normalizations applied to documents before they are indexed.
- **Query preprocessing** – the enhancements and normalizations applied to the query before it is used for retrieving and ranking the documents.

Summary

To summarize, we note that this scenario includes three tasks, each of which can be modeled individually using a processing framework. However, with the present processing frameworks, they cannot be modeled as a single workflow. We also note that the experiment calls for parameter sweeping to try different configurations for the ranking function, for the preprocessing of documents and queries, and to run the experiment on different data sets. Depending on the parametrization, the different analysis components are involved in the preprocessing of documents and queries, so these preprocessing workflows need to be assembled dynamically. However, parameter sweeping and the dynamic assembly of workflows are also not readily support by the processing frameworks. Our approach, as implemented in DKPro Lab, provides a lightweight and non-invasive way of integrating the workflows into a single comprehensive experimental setup, while at the same time adding support for parameter sweeping and the ability to dynamically change the structure of the workflow depending on its parametrization. In this way, the scenario, which previously required the manual reconfiguration of the involved components and the manual execution of each involved workflow in turn, can be run fully automatically.

4.2.4.2 Scenario 2: Machine learning

Machine learning (ML) approaches can be applied to a variety of language analysis tasks, such as part-of-speech tagging or named entity identification. Based on a manually annotated corpus, a classifier is trained. The classifier works well if it can correctly assign the categories it was trained on to unseen data. DKPro Lab is used to implement such a machine-learning use-case in the DKPro Text Classification [65] project. DKPro-TC supports parameter-sweeping experiments which integrate preprocessing, training, and cross-validating of classifiers into a single experimental workflow.

Machine learning frameworks like WEKA [110] typically also offer support for training and cross-validation, but expect that the data already has been preprocessed. Our approach provides a way to integrate preprocessing, training, and cross-validating of classifiers into a one parametrizable experimental setup which can be run fully automatically.

ClearTK-ML also supports the integrated preprocessing, training, and cross-validating of classifiers. However, it does not provide the ability to perform these steps in conjunction with a parameter sweeping, which the combination of DKPro TC and DKPro Lab can provide.

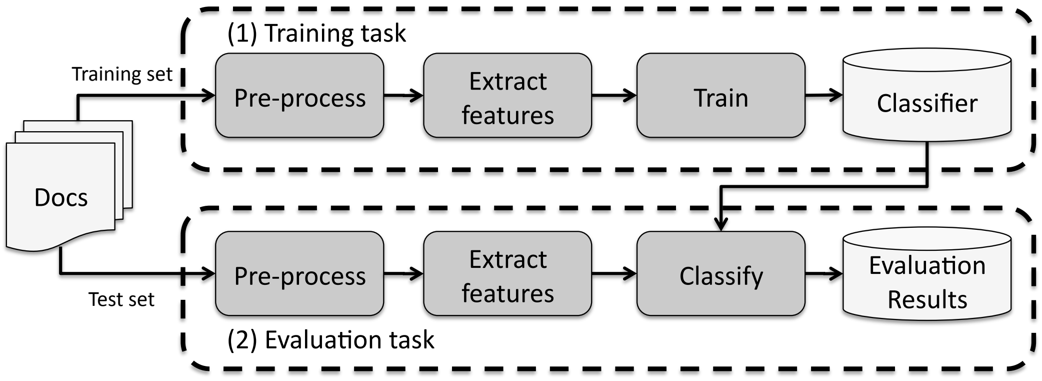


Figure 4.10: Machine learning experiment workflow

Workflow structure

An experimental setup consists of two tasks: 1) training the classifier based on feature extracted from the data, and 2) evaluating how the classifier performs on unseen data (Figure 4.10). Since the preprocessing and feature extraction is exactly the same in both tasks, they could also be modeled as separate tasks.

Parameters

There is a wide variety of possibilities for parametrization in an ML experiment. The manually annotated data can be enriched with additional information before the feature extraction takes place. Many features can be extracted from the data. The features can be used in conjunction with different ML algorithms to train a classifier. To summarize, the ML experiment roughly includes the following parameters:

- **Data set** – a set of annotated documents, which can be split into a training and test sets.
- **Document preprocessing** – the enhancements and normalizations applied to documents before the features are extracted.
- **Feature extraction** – the extraction of features from the preprocessed documents which can be used to train a classifier for the automatic annotation of documents according to the categories annotated in the data set.
- **Algorithm** – the machine learning algorithm used to train a classifier from the extracted features. Effectively, this parameter controls which machine learning library to use (such as WEKA [110] or Mallet [150]), how the extracted features need to be encoded (e.g. as a WEKA ARFF file), and which algorithm provided by that library is then invoked.

Cross-validation

A special kind of parameter variation is required for cross-validation. Cross-validation is a technique to help the researcher to determine the quality of a classifier without the need for a separate test set. It is often used during the development of new approaches in order to test experimental configurations. A part of the training data is separated, the classifier is trained using the remaining data and evaluated against the previously separated data. If we assume that the set of data instances used to train the classifier is one parameter, a cross-validation scenario requires some way of expressing that a workflow should be repeated several times, every time using a different split of the data instances into training and test sets. Thus, cross-validation requires one parameter, the documents in the data set, to be dynamically split into two new parameters, the documents in the training set and the documents in the test set.

Summary

Experimenting with different preprocessing steps, feature extraction strategies, and machine learning algorithms, makes machine learning experiments a natural target for parameter sweeping and dynamically assembled workflows. Using our approach, cross-validation experiments can easily be modeled at the level of the parameter space, making it a natural task to perform instead of a special concept the framework needs to support. This makes the DKPro Lab framework a natural choice as a basis for the DKPro TC framework. Additionally, the programmatic approach and the ability to dynamically assemble workflows that are provided by DKPro Lab enable the DKPro TC framework to provide extra convenience to the user, e.g. to dynamically add preprocessing steps depending on the kinds of feature extractors being used.

4.2.5 Summary

In this section, we have presented an approach to analysis workflows that change their structure dynamically based on their parametrization, e.g. in parameter sweeping experiments where the same setup is run with many parameter settings. Such a parameter variation can be used to find the optimal configuration for an experimental setup.

However, as Drummond [68] points out, simply repeating an experiment is not sufficient to validate the underlying hypothesis. He suggests that variation of the whole experimental setup is important. In our approach, parameters can affect the actual structure and make-up of a workflow, which facilitates incorporating such structural variations directly into the experimental setup.

An important goal of our approach and its implementation in the DKPro Lab framework [72] is to be lightweight and unintrusive. Existing code can be easily wrapped to run within the framework. Users can adopt the framework gradually at their own speed and specialize it to their problem domain, e.g. by implementing custom task categories, parameter dimensions, or reports. Unlike other workflow systems, our framework focuses explicitly on the programmatic creation of complex workflows and thus on building upon existing programming skills and tools for the Java platform. This facilitates the creation of experimental workflows, their debugging, and refactoring. In particular, debugging is not only limited to the level of the workflow, but can seamlessly go down to the level of individual analysis components or even further down to the supporting libraries.

By making the parameter sweeping workflow independent of the processing framework, we can accommodate experimentation across multiple processing frameworks and incorporate arbitrary processing steps. This facilitates the automatization of auxiliary steps, typically performed manually and therefore being prone to errors, e.g. to direct output of one step of the experiment into the next one. Exploiting the data dependencies between processing steps allows the framework to smartly run processing steps only when necessary and to automatically reuse intermediate data.

We designed our approach to facilitate the building and sharing of complex experimental setups. In fact, we can report that the DKPro Lab framework is already being used in diverse research tasks, e.g. by Ferschke et al. [85] in a system for predicting quality flaws in Wikipedia, by Daxenberger and Gurevych [56] in experiments on automatically classifying edit categories in Wikipedia revisions, by Flekova and Gurevych [87] for a large scale age and gender author profiling study in social media, by Zesch and Haase [234] for research on preposition and determiner correction, and by Zesch et al. [236] in a system using text similarity metrics for the textual entailment task. Some of these researchers have made their experimental setups publicly available, e.g. Zesch et al. [236] as part of DKPro Spelling [64].

We also had the goal that users would adopt the concept of reusable task categories and reports and that they would provide specializations of the DKPro Lab framework for new problem domains. The DKPro Text Classification framework [65] is such a new specialization for text classification based on machine learning. It emerged as a joint effort from previous work done by Ferschke et al. [85], Zesch and Haase [234], and Daxenberger and Gurevych [56], and provides new task categories and reports specific to this text classification and machine learning experiments.

Future work is possible in various directions. One direction is an extension of the simple parameter sweeping support towards an optimizing sweep which prunes unpromising parameter combinations from the parameter space. This would allow the exploration of larger parameter spaces. Another direction is the support for compute clusters, e.g. via Apache Hadoop [8] and DKPro BigData [61]. The recently published CSE Framework by Garduno et al. [97] may serve as an inspiration for such work. It supports parameter space pruning and the deployment on a compute cluster. However, unlike our approach, it is tightly integrated with the UIMA framework.

5 Flexibility

Users have different goals and requirements when analyzing language. Some users search for linguistically interesting grammatical constructions, others want to improve the information retrieval of search engines, or perform information extraction on resumes to help human resource managers in staffing. There is a great variety of scenarios in which language analysis plays an important role. To avoid building analysis systems fully from scratch every time, flexible data structures, commonly used annotation type systems, and interoperable analysis components are required.

Flexibility of data structures

To provide for maximum flexibility, processing frameworks keep the data structures used to model analysis results very generic. While some older systems (e.g. [154; 41]) were built around XML and focused on trees, and later on ordered directed acyclic graphs, current systems support arbitrary graphs. The graphs are not necessarily explicitly modeled as nodes and edges, but rather a graph of feature structures in which a node represents a set of key/value pairs. An edge in such a graph is represented as a feature value referencing another node.

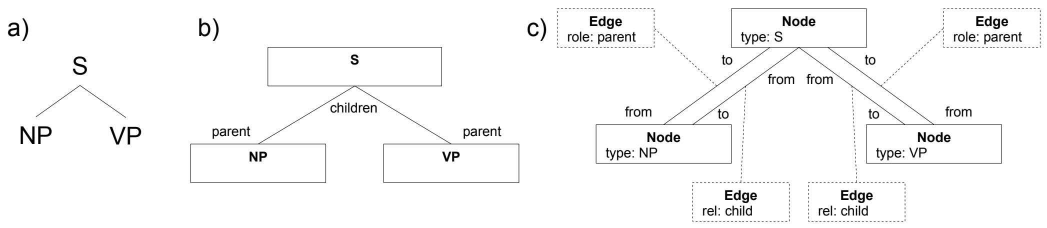


Figure 5.1: Representations of a tree: (a) Tree fragment represented as (b) feature structure graph with edges represented as feature values referencing another feature structure and as an (c) explicit graph with nodes and edges.

The data structures used for the communication between analysis components are defined by the processing framework. However, the annotation type system and the analysis components need to be defined by the user and specialized for the analysis task at hand.

A type system defines how information can be modeled and stored in the data structures. The type of a node determines which features are available. The type of a feature determines which values it may assume. Consider a type *Token* represented by key/value pairs with the keys *begin*, *end* (integer character offsets), and *partOfSpeech* (string). It is often possible to create derived types, e.g. a type *TokenWithLemma*, which inherits all features from its supertype *Token* and adds a new feature *lemma* (Figure 5.2). This corresponds to the inheritance mechanism that can be found in many object-oriented programming languages.

For automatic annotation, the type system assumes the function of an interface specification between the analysis components. One component stores its analysis results as objects of a well-known type, while another component can collect them from there.

For manual annotation, a type system can become part of the annotation guidelines and thus of the collaboration agreement between annotators. In order to compare the annotations from

Definition: *feature structures* – A feature structure (FS) is a typed container for key-value pairs. The keys are strings representing the names of the features. The values are either primitive (e.g. numeric, string, etc.) or a reference to another FS.

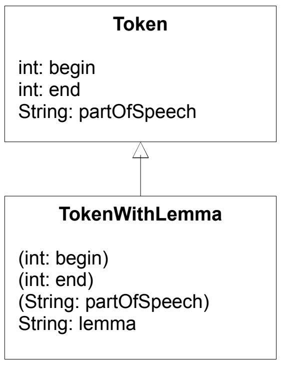


Figure 5.2: Types and inheritance: the type *TokenWithLemma* is derived from the type *Token*

different human annotators, it is important that they represent their analysis in the same way. This comparison is the basis for calculating the inter-annotator agreement (cf. [40]), which is an important instrument in assessing the quality of manually created annotations.

Flexibility of processing

The ability to integrate arbitrary kinds of analysis processes is already provided by flexible data structures and type systems. However, the ability to use components interchangeably, i.e. to replace an analysis component of a certain kind with another component of the same kind, is not immediately provided by processing frameworks, but rather it is a trait of curated collections of analysis components. For example, consider an analysis workflow using a slow high-quality parser is about to be embedded in an interactive application. However, for the application, quality is less important than speed. To operate well in this scenario, the parser is replaced with a faster, less accurate implementation. There is a variety of other reasons why components may be replaced in an analysis workflow.

From the perspective of the user, it is desirable that the change of a component is a local operation. I.e. the change should not require further, potentially extensive changes throughout the whole analysis workflow. It should also be a minimal change. This requires a high degree of homogeneity between the components, as only a well-curated collection of analysis components can provide it.

Manual vs. automatic analysis

Manual and automatic analysis tasks have contradictory requirements towards flexibility. A well-known type system enables the interoperability of analysis components and therefore is essential for providing a collection of reusable analysis components. At the same time, defining a type system limits the freedom of a human annotator in a manual analysis task, in particular during phases of an annotation project where the analysis guidelines have not been finalized yet.

In Section 5.1, we compare different annotation type systems to get a better understanding of the decisions that their designers take and their consequences for manual and automatic annotation. In Section 5.2, we discuss the challenges of building a large collection of interoperable analysis components.

5.1 Annotation type systems

In this section, we identify patterns underlying the design of annotation type systems and analyze their relation to each other. We compare different type systems to derive recommendations that can be used for the design of new annotation types or new type systems. We discuss whether there are sufficient similarities between the type systems to warrant an attempt to create a common type system which could be used by multiple component collections. Additionally, the knowledge about common design patterns can help the developers of annotation tools to provide user interfaces and interactions tailored to these common patterns. Finally, we consider whether the specification of the type systems is sufficient for manual and automatic analysis tasks, or if additional specifications are required.

This section addresses the following issues in our overall scenario (Figure 5.3):

❸ **Automatic analysis tools and annotation editors are not interoperable.**
Our analysis provides developers of annotation editors with a set of patterns to expect in type systems. They can then offer user interfaces tailored specifically to these common patterns and offer a better interoperability with type systems based on these designs.

❼ **In automatic analysis, annotation type systems are predefined, but manual annotation requires customizability.**
We conclude that a compromise can be reached by limiting the freedom to define own types in annotation editors to certain type system design patterns. E.g. tag sets could be customizable, while the design decision for representing labels is predetermined. Additionally, constraints are needed to prevent users from creating malformed annotations.

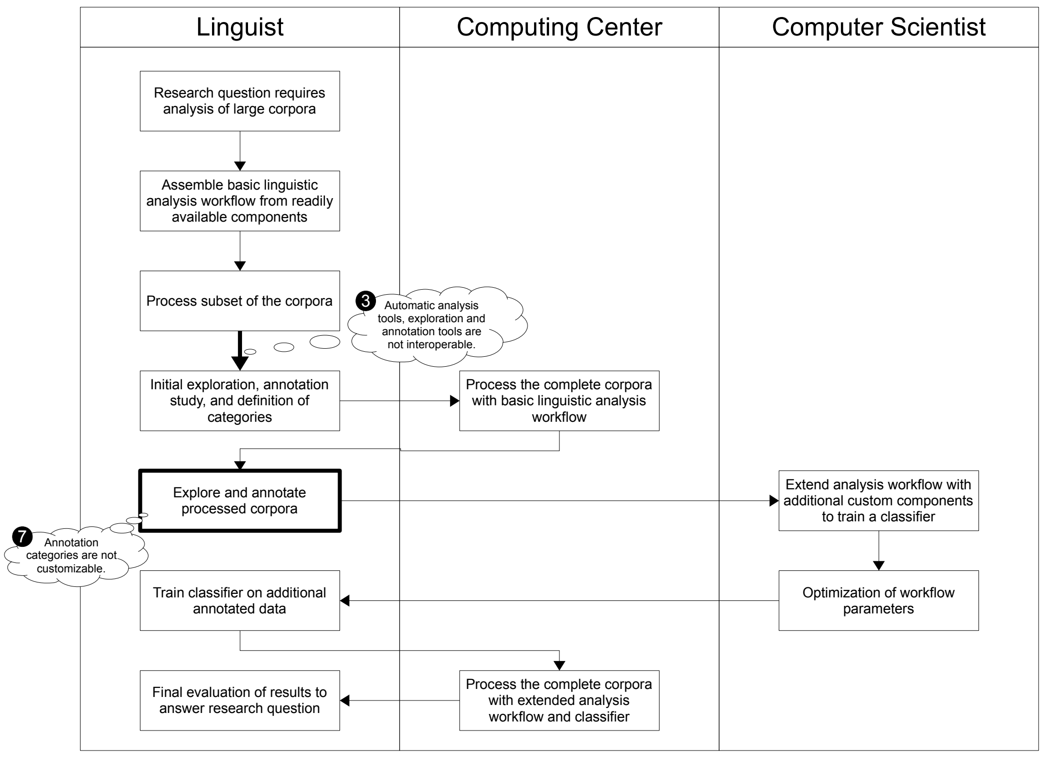

Linguist
Computing Center
Computer Scientist
Research question requires analysis of large corpora
Assemble basic linguistic analysis workflow from readily available components
Process subset of the corpora
❸ Automatic analysis tools, exploration and annotation tools are not interoperable.
Initial exploration, annotation study, and definition of categories
Process the complete corpora with basic linguistic analysis workflow
Explore and annotate processed corpora
Extend analysis workflow with additional custom components to train a classifier
❼ Annotation categories are not customizable.
Train classifier on additional annotated data
Optimization of workflow parameters
Final evaluation of results to answer research question
Process the complete corpora with extended analysis workflow and classifier

Figure 5.3: Design patterns in annotation type systems can guide type system designers to avoid a proliferation of incompatible type systems. Tool designers can focus on supporting these designs, in particular when allowing users to create custom types.

5.1.1 Motivation

The representation of linguistic information has been studied extensively in the past. Among others, Chiarcos et al. [44] refer to three levels of representation formats, the *physical*, the *logical*, and the *conceptual* level. The physical level refers to the serialization format used to persist or exchange annotated data (e.g. XML, RDF, PAULA [44], GrAF [118], or XMI [173]), the logical level refers to the abstract data model being used (e.g. annotation graphs [25], heterogeneous relation graphs [213], the PAULA object model [44], LAF [117], or the UIMA CAS [105]), and the conceptual level refers to the annotation types (see below) and tag sets (cf. ISOcat [130], OLiA [43], and Section 5.2.4.2). The abstract data model is also referred to as the *meta model*, whereas the conceptual data model is referred to simply as the *data model*.

As the eco-system around the UIMA framework is steadily growing, we take the opportunity to analyze how different type systems based on the meta model of the UIMA framework have been designed. To the best of our knowledge, this is the first comparative design study of different type systems using the UIMA meta model. We focus on analyzing type systems based on the UIMA meta model for two reasons:

1. **Comparability** – Design decisions can be directly compared to each other. If we had instead compared the representation of a corpus in, for example, the PAULA meta model [44] and the annotation graph meta model [25], the design decisions would be heavily influenced by the meta model itself. However, we were interested in the design decisions that have been taken by different parties that operate on the same meta model which is based on a feature structure graph. As even within the same meta model, very different designs can be found, we believe this to be a valuable contribution.

2. **Potential for consolidation** – Every major analysis component collection for UIMA currently defines its own type system (ClearTK [172], cTAKES [190], DKPro Core [Section 5.2], JCoRe [109], and U-Compare [129]). Thus, there are currently several competing type systems. By comparing these type systems, we wish to assess the potential for setting up a common best practice type system, which could be used by multiple analysis component collections and, thus, foster interoperability and reduce the duplication of work. From personal communication with developers of these collections, particularly from the DKPro Core, cTAKES, and ClearTK communities, we know that there is interest in a common type system. However, it is yet unclear how such a type system should be designed. The present work represents a step towards the proposal of a common type system.

An annotation type system is the machine-readable counterpart to the annotation guidelines given to human annotators. In fact, when configuring an annotation editor for a particular annotation project, relevant parts of the annotation guidelines may be extracted and converted to a type system, which the editor uses to guide and assist the manual annotation process. But as an annotation type system also serves as part of the interface specification between analysis components, it becomes part of their API. As a result, type systems can be seen as an interface specification between the manual and the automatic annotation processes.

Best practices

It is commonly known that the development of new annotation types and type systems is not a simple task. Domain-specific needs are to be balanced against practical and technical considerations. There have been various publications (e.g. [190; 109; 129]) on the subject, each proposing a different design. Also, researchers that do not spend their time on analyzing and comparing existing approaches, define types ad-hoc, which may later require significant refactoring when requirements appear which had been initially unknown.

By examining and comparing different type system designs, we can highlight common design decisions and discuss their respective benefits and drawbacks. The results of our analysis can help developers of new types and type systems to make informed decisions.

Type systems for manual annotation

For an efficient manual annotation process, annotation editors need to provide good visualization and interaction modes depending on the information, or rather on the specific kind of information that is annotated. Our analysis of the design patterns used in different type systems improves our understanding of the inventory of visualization and interaction modes that an annotation should provide in order to support these type systems. More specifically, if the editors are intended to support the definition of custom annotation types, it helps to understand which differences to expect between the data model used for visualization and the data model used for representing analysis results. In this respect, the analysis performed here serves as a preparatory study for a new extension of the state-of-the-art annotation editor WebAnno [232], allowing for the definition of custom annotation types.

Towards a common type system for automatic analysis

It is important to think about a common annotation type system that is used by multiple component collections and permits extensive interoperability between analysis components. In particular at the lower analysis levels, much work is duplicated between the different component collections. There are many tokenizers, sentence boundary detectors, part-of-speech taggers, or lemmatizers repeatedly wrapped as UIMA analysis components. Agreeing on a common representation, even for the most basic types, would immediately allow avoiding duplicate work and potentially to increase the coverage of languages and domains for which interoperable components are available. Differentiating properties between component collections would still remain, e.g. by focusing on certain languages or domains, by focusing on portability, speed, or other useful properties. Our analysis helps us understand if it is feasible to design such a common type system, and how to approach it.

5.1.2 State of the art

To determine common design decisions, but also different approaches, we compare the type systems for linguistic annotations from five different analysis component collections based on the UIMA framework. As the UIMA framework does not provide any domain-specific annotation types and since, at least so far, there is no commonly agreed-upon type system for linguistic analysis, each component collection uses its own independent type system. The collections examined later in this section are:

- ClearTK [172]
- cTAKES [190]
- DKPro Core [Section 5.2]
- JCoRe [109]
- U-Compare [129]

Some of these type systems contain very specific types for certain domains, e.g. cTAKES has a strong focus on the medical domain. However, all type systems also cover linguistic concepts, such as tokens, sentences, syntactic constituency structures, dependency relations, etc. In our comparison, we focus on the representation of these common concepts, because this is where the potential for consolidation can be found.

In the rest of this section, we use the following nomenclature from the UIMA framework. A feature structure (FS) is a typed container for key-value pairs. The keys are strings representing the names of the features. The values are either primitive (e.g. numeric, string, etc.) or a reference to another FS. An *annotation* is an FS which is anchored to the text, bearing two well known numeric features, *begin* and *end*, representing offsets in the annotated data. In UIMA, all primary data and FSes are stored in a Common Analysis System (*CAS*) [105] object, which is used for exchanging data between analysis components.

During our analysis, special attention is paid to the benefits and drawbacks these patterns have, not only with respect to automatic processing, but also with respect to manual annotation. For example, some annotations types are closely related to each other and form a conceptual *layer*. Annotation editors are tools for manually analyzing language data and annotating it, e.g. the UIMA Annotation Editor [13] or WebAnno [232]. To avoid overloading the visual appearance, such editors support the ability to show only certain types of annotations. For a user, it is more convenient to use such a functionality on the level of the conceptual layer, e.g. than applying it to each annotation type individually.

Applicability to other processing frameworks

Although we focus on UIMA here, the design patterns apply to other frameworks as well. UIMA type systems can be compared to an object-oriented system with single inheritance (cf. [210]), i.e. a type can only have a single supertype. In fact, UIMA provides a code-generation module which renders types defined in a UIMA type system as Java classes, employing getters and setters for feature values and inheritance, but not interfaces. Additional custom methods can be added to these generated classes. Tesla directly employs the Java type system as its annotation type system and defines a set of Java classes that are used to store analysis results. Additional design options could be introduced through the added support for interfaces, which we would miss in our analysis. Unlike Tesla and UIMA, GATE does not require the definition of types. Annotations do not have a particular type, but are represented as generic key/value sets, like the UIMA FSes. Since large parts of our analysis are related to how to model relations between FSes, without explicitly taking named types into account, they are relevant for example for GATE as well.

We now briefly introduce the component collections whose type systems we compare in this section.

5.1.2.1 ClearTK

The *ClearTK* analysis component collection [172] has a strong focus on machine learning, providing special support for feature extraction and for different machine learning frameworks. Based on these, custom analysis components for all kinds of tasks can be implemented, provided that annotated data exists from which the machine learning algorithms can be trained. Finally, ClearTK integrates several analysis components wrapping third-party tools, such as part-of-speech taggers, parsers, etc., which mainly serve to preprocess the data before the feature extraction is performed. Further details regarding this component collection are provided in Section 5.2.

5.1.2.2 cTAKES

The *clinical Text Analysis and Knowledge Extraction System* (*cTAKES*) [190] is a project focusing on the automatic analysis of medical records. It consists mainly of a collection of analysis components specifically targeted at the application domain, e.g. annotation of drugs, finding mentions of side effects of drugs, etc. For linguistic preprocessing tasks such as part-of-speech tagging, chunking, or parsing, cTAKES incorporates one analysis component for each task. The cTAKES system provides basically a single, comprehensive analysis workflow in which the components and resources are tuned to interoperate optimally. The *SHARPn common type system* adopted by the cTAKES project since version 2.5 is described by Wu et al. [227]. Further details regarding this component collection are provided in Section 5.2.

5.1.2.3 DKPro Core

DKPro Core (Section 5.2) is a part of the *Darmstadt Knowledge Processing Repository* [107] and focuses on basic linguistic preprocessing tasks, covering primarily segmentation, part-of-speech tagging, lemmatizing, syntactic parsing, dependency parsing, named entity recognition, and coreference resolution. DKPro Core serves as an integration framework for many third-party tools which handle a wide range of analysis tasks. It is the mission of the project to provide the user with a rich choice of appropriate tools for each of the tasks.

5.1.2.4 JCoRe

JCoRe [109] is a comparatively small collection of analysis components. Contrary to the other collections, it is not advertised and versioned as a single package, but rather each component is advertised separately. All components are based on the JCoRe type system, also known as the JULIE Lab type system, which has been the topic of several publications [38; 108]. Unlike other type systems, the JCoRe type system does not primarily aim at a high-level interoperability between components, but also contains specialized types for particular corpora (e.g. a syntactic constituent type, *PTBConstituent*, for the Penn Treebank [146]), for shared tasks (e.g. the type for the *Automatic Content Extraction (*ACE*) Program* [66]), or for tools (e.g. MMAX2, [164]).

5.1.2.5 U-Compare

U-Compare [129] is a system to compare the results produced by different analysis components or analysis workflows with each other. It consists of a workflow editor and integrates a wide variety of analysis components, which often call out to third-party web services. In addition to annotation types which enable the interoperability between analysis components, the U-Compare type system provides a set of types dedicated to the encapsulation of results produced by individual analysis components which are used in the comparison process. The U-Compare type system was described in detail by Kano et al. [128]. Further details regarding this component collection are provided in Section 5.2.

5.1.3 Contribution: An analysis of type system designs

Examining the aforementioned type systems, we encounter several strategies of building a type system. In this section, we describe these strategies and discuss their respective benefits and drawbacks.

5.1.3.1 Structural patterns

Annotations form structures over the annotated data. The most common structure is a *span*, a section of the annotated data delimited by *begin* and *end* offsets (e.g. used for tokens, sentences,

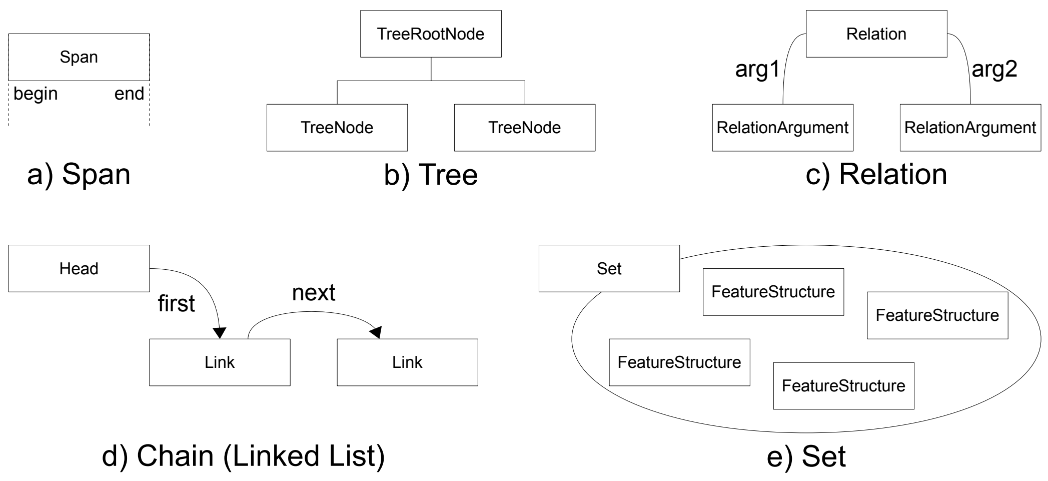


Figure 5.4: Common structures for annotation

etc.). There are other common structures, such as *trees* (e.g. syntax trees), *linked lists* (e.g. coreference chains), *relations* (e.g. dependency relations), *sets* (e.g. named entity mentions).

Structures can be realized as structural base types, which implement them in a generic way, without any relation to a linguistic theory or other domain, such as medicine. Domain-specific types are derived from these generic types, e.g. types for the annotation of dependency relations could be derived from structural base types for relations. (Figure 5.4).

Alternatively, structures can be realized directly as domain-specific types. E.g. a type *Constituent* with features *parent* and *children* can be used to represent nodes in a constituency parse tree without being derived from a more generic *TreeNode* type.

Span [Figure 5.4 a)]

The span is the simplest and most common annotation structure. A span annotation simply marks a continuous region of the primary data, specified by begin and end offsets. Spans are often a built-in type in processing frameworks and special functionality is provided to handle spans, e.g. efficiently fetching all other annotations within a span, fetching the portion of the primary data marked by the span, etc.

In the UIMA framework, the span is represented by the built-in type *Annotation*. All the examined type systems make heavy use of this structural type and most of their types are derived from *Annotation*. The type defines two features *begin* and *end* which represent character offsets in the annotated text data.

Tree [Figure 5.4 b)]

Trees are primarily used to model the syntactic constituent structure in a sentence. They could also be used to model other annotations, e.g. the document structure (chapter, section, paragraph, etc.).

Explicit modeling of the tree structure is not necessary in case when the nesting information is implicitly encoded otherwise. For example, in the case of the document structure, it can be

Definition: *structural base types* – Types used to represent a structure in a generic way, without any relation to a linguistic theory or other domain, such as medicine. Domain-specific types are derived from these generic types, e.g. types for the annotation of dependency relations (e.g. *DependencyRelation*) could be derived from structural base types for relations (e.g. *Relation*).

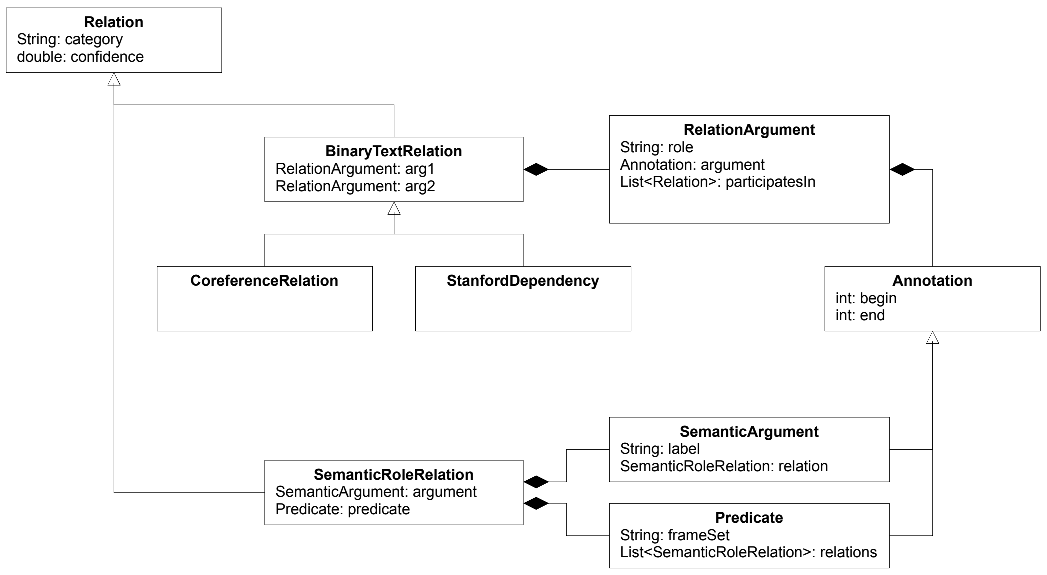


Figure 5.5: Relation types in cTAKES (excerpt)

assumed that all sections of a document are properly nested in each other, i.e. that no two sections have the same offsets and all sections are fully contained within another section. In that case the tree structure can be fully obtained from the containment information. For the case that containment is ambiguous, because two annotations have the same offsets, UIMA provides the *type priority* mechanism, defining the relative stacking of annotations. For example, if a heading is made up of a single token, the priority may define that *Heading* has a higher priority than *Token*, meaning that the heading contains the token, and not vice versa.

The U-Compare type system provides a structural type *TreeNode*, which is used as a base type for further types representing the syntactic constituency structure (*Constituent*). The type system also provides a refined set of types for representing the document structure. Even though the document structure modelled as a tree is explicitly encoded here, the *TreeNode* is not used as a base type for the document structure types.

The other type systems do not provide an abstract type for trees. However, each type system declares types for representing the syntactic constituency structure. DKPro Core, ClearTK, and cTAKES declare dedicated types for the root node of the tree. ClearTK and cTAKES additionally declare dedicated types for the leaf nodes of the constituency tree. In DKPro Core, the leaves of the constituency tree are *Tokens*.

Relation [Figure 5.4 c)]

Relations are a very versatile annotation structure. They are mostly used to model dependency relations, but can also be used to model coreference relations, temporal relations, etc. None of the examined type systems, except cTAKES, provides a base type for relations. The UIMA meta model represents a graph structure, with feature structures being the nodes and complexfeatures being the edges. However, these edges cannot have labels (features) themselves. A relation can be seen as a way of modeling an edge as a proper feature structure, so that is can bear features itself.

The cTAKES type system provides a generic *Relation* type which serves as the basis for a whole set of specialized types (cf. Figure 5.5). This type is not anchored to text, i.e. it does not bear *begin* and *end* features and it is not derived from the built-in *Annotation* type. Neither does the relation base type specify any arguments, such as *arg1* and *arg2*, which are only introduced in more specific relation types such as *BinaryTextRelation* or *ElementRelation*. Instead, it bears features such as *category* and *confidence*.

This design allows specialized relations to use descriptive feature names, e.g. the *SemanticRoleRelation* type bears the features *predicate* and *argument* instead of *arg1* and *arg2*. On the other hand, the design also disqualifies *Relation* as a structural base type, because it bears no structural information, such as references to the relation end points.

However, the *BinaryTextRelation* could be considered a structural type. It defines the features *arg1* and *arg2* and serves as the base for types such as *CoreferenceRelation* and *StanfordDependency*. As a consequence, these types do not have descriptive names for their relation features, unlike *SemanticRoleRelation*. Instead of being encoded as feature names, the kind of relation is encoded in the *role* feature of the *RelationArgument* type.

It is unclear why the *SemanticRoleRelation*, which obviously is a binary relation on text segments, has not been implemented as a subtype of *BinaryTextRelation*. However, the comparison of these types nicely highlights the benefits and drawbacks of a structural type (here *BinaryTextRelation*). The use of the structural type introduces additional complexity (here the *RelationArgument* type) and inconvenience, e.g. non-descriptive feature names. However, it provides a uniform way to access different kinds of relations (here the *CoreferenceRelation* and the *StanfordDependency*). On the other hand, the *SemanticRoleRelation* and its associated types bear descriptive type and feature names, but are not accessible in the same uniform way as the relation types derived from the *BinaryTextRelation* types.

Some type systems anchor relation types directly to the text with *begin* and *end* offsets, e.g. the DKPro Core and U-Compare type systems, others do not, e.g. the ClearTK and cTAKES type system. Instead, the latter anchor relations indirectly to the text, e.g. through the end points of the relation (cf. Figure 5.5). However, the access and indexing mechanisms provided by the UIMA framework assume that annotations are directly anchored on the text. For this reason, getting a list of relations in the order they appear in the document, or getting all the relations pertaining to a particular section of the document, e.g. all dependency relations for a sentence, is neither easy nor efficient. Thus, a component collection providing indirectly anchored annotations should also provide a library of functions dedicated at accessing and handling these. This is also particularly inconvenient for tools which provide editing or visualization capabilities, because they cannot rely on the default assumption that annotations are directly anchored, but need to handle the specific approach to indirect anchoring.

Linked List [Figure 5.4 d)]

There is a linked list built directly into the UIMA framework: the type *FSList* and its subtypes (Figure 5.6). The type *NonEmptyFSList* represents a link in the list and has two features: *head* pointing at the FS associated with the link and *tail* pointing at the next link. The *FSList* types do not represent structural base types, because it is not possible to derive specialized kinds of linked lists, e.g. for coreference chains. Instead, the *FSList* serves to provide features with multiple values, in particular when the number of values is not previously known.[1]

Whereas the DKPro Core type system does not provide a structural type for linked list, it uses a linked list structure to model coreference chains. The type *CoreferenceChain* resembles the head of the list and provides a pointer to the first element. The *CoreferenceLink* type is used to model the elements of the list. Being derived from the *Annotation* type, they are anchored to the text. The feature *referenceType* specifies the role of the link itself within the chain. The feature *referenceRelation* qualifies the relationship to the next element in the chain pointed to by the feature *next*. Conceptually, *referenceRelation* is a label on the edge to the next element.

The other type systems do not employ types which form linked list structures.

[1] If the final number of values is known, *FSArray* is the better alternative for multi-valued features.

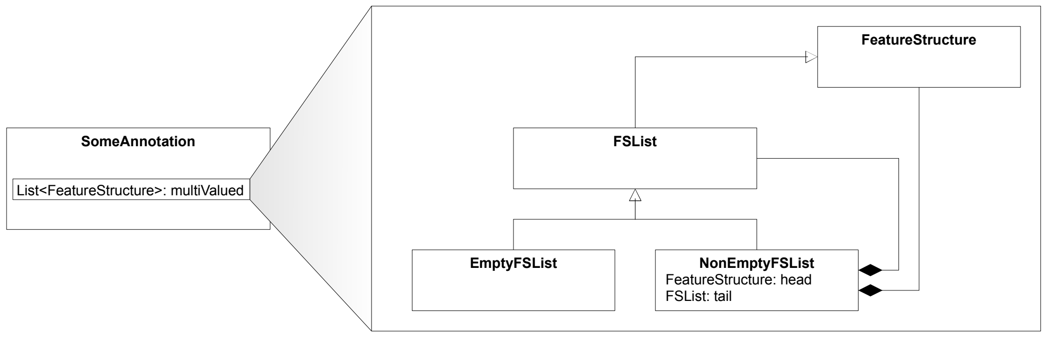


Figure 5.6: UIMA *FSList* type hierarchy

Set [Figure 5.4 e)]

A *Set* groups feature structures that share a certain quality. The shared information is represented by features on the *Set* type. The members of the set are referenced by a multi-valued feature (e.g. *FSList* or *FSArray*).

This structure is often used to model abstract named entities and their mentions in the text, which is one of the ways of expressing coreference. The U-Compare type system provides the *LinkedAnnotationType* to model a set of annotations. This type is used as a base type for the *NamedEntity* type. The set represents the entity itself, while members of the set correspond to the mentions of the entity. cTAKES provides a similar type called *CollectionTextRelation* used to model coreference relations. The U-Compare type system also uses a set type (*LinkedAnnotationSet*) to its representation of named entity annotations.

Discussion

Should generic structural base types be used, or should structures be only implicitly realized in domain-specific types (e.g. types for constituency structure, coreference chains, etc.), and what are the positive or possibly negative consequences?

Structural types are base types from which the more specific types, such as *Token* or *Constituent* are derived. The derived type inherits all features from its parent, which of course is desired, but can also have a negative effect. In particular, the derived type might require more specific constraints on the features. While a generic *Relation* type may have two features *arg1* and *arg2* of the type *Annotation*, the derived type *DependencyRelation* should be constrained to form relations between *Tokens*.

The naming of features is another problem. Consider again a *DependencyRelation* type. Dependency relations exist between a *governor* and a *dependent*, but the names of the features indicating the end points of the relation are already called *arg1* and *arg2* in the base type *Relation*. Unfortunately, there is no way to rename features, e.g. *arg1* to *governor*.

However, structural types are very useful if there are applications which are agnostic of the semantics of a type, e.g. if a relation is a dependency relation or some other kind of relation. For example, to visualize a structure, to choose an efficient strategy to index a structure in a database, or possibly to convert data from one representation format into another, it is often sufficient to know the basic structures. If this information is not available from the structural base types, an external mapping would be required, e.g. indicating that a *DependencyRelation* is a binary relation and that the end points are specified by the *governor* and *dependent* features.

For the DKPro Core type system, we chose not to use structural types to avoid the problem of naming features. We aimed to retain the ability of giving features descriptive names according to their actual roles in the annotation, providing the user with a better intuition what these features stand for without having to consult documentation.

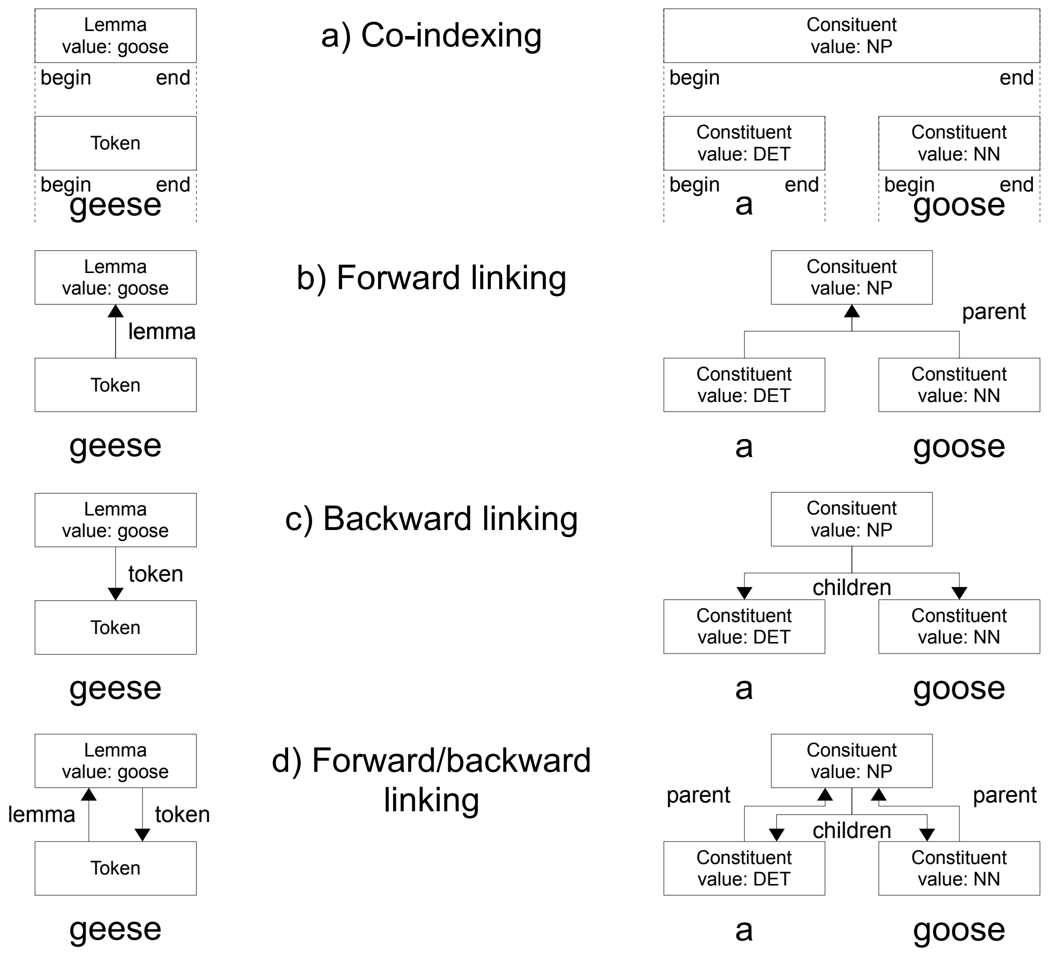


Figure 5.7: Strategies to associate annotations/feature structures with each other

5.1.3.2 Association patterns

Annotations do not stand alone, they relate to each other and to the primary data. The part-of-speech tag or lemma relate to a token, because they are generated specifically for this token and based on its context. A token relates to a sentence, because it is situated inside the sentence boundaries, usually representing a word within the sentence. There are several ways to express such relationships, e.g. explicitly by various ways of setting up links between related annotations, or implicitly by using co-indexing.

Co-indexing [Figure 5.7 a)]

Co-indexing is an implicit method of expressing a relationship between annotations. By convention, we can say that two annotations of a certain kind relate to each other, if they share the same position in the document (start and end offsets are equal). Since most types in the examined type systems are derived from the UIMA *Annotation* type, most of them can be implicitly associated via co-indexing.

Forward-linking [Figure 5.7 b)]

For the linking strategies, we assume that there is some kind of order or direction in the relationship. E.g. in order to determine the lemma for a token, the token has to be known first. So the annotation specifying the token exists before the annotation resembling the lemma is made. A forward-linking relation is one from the earlier, or lower-level annotation to the later, or higher-level annotation.

If the annotations are in a hierarchical or dominance relation, e.g. the child-parent relation in a tree, this would be the relation from child to parent.[2]

Forward linking is commonly used to link types that are anchored to the primary data to labels, because it often corresponds to the typical navigation path. E.g. DKPro Core and JCoRe use forward linking to associated tokens with lemmata, part-of-speech tags, and stemmed forms.

A linking association is not an FS by itself. It is rather expressed by a feature on one FS whose value is a pointer to the other FS. So an FS of the type *Token* can have a feature *lemma* pointing to another FS of the type *Lemma*.

Backward-linking [Figure 5.7 c)]

A forward-linking relation is one from the later, or higher-level annotation to the earlier, or lower-level annotation. As said before, it is necessary to know the token before its lemma can be determined. In a backward-linking scenario, the *Lemma* annotation would have a feature *token* pointing to the *Token* annotation for which the lemma was generated.

If the annotations are in a hierarchical or dominance relation, e.g. the child-parent relation in a tree, this would be the relation from parent to child.

Across all type systems, backward linking is commonly used for more abstract annotations such as named entity mentions, dependency relations, or coreference relations.

Forward-backward-linking [Figure 5.7 d)]

Forward-backward-linking is a combination of the two previously addressed linking strategies. Both related annotations carry a feature linking to the respective other annotation. With exception of the JCoRe type system, forward-backward-linking is used to model the parent/child relationship in constituency trees. The cTAKES type system also uses this strategy to associate relations with their arguments.

Type merging

The ability to define a type by the same name multiple times and merge these definitions is a special concept in the UIMA framework. Consider the type *Token* being defined as a simple annotation only bearing *start* and *end* offset features. A second definition of the type includes a *lemma* feature and a third one includes a *pos* feature. When an analysis workflow is initialized with all three definitions, they are merged into a final type definition.

Type merging can be used to create additional features on types originally defined in different modules without resorting to inheritance. However, it is not used by any of the examined type systems. It also does not interact well with another concept in UIMA, the *JCas* [193]. While UIMA type systems are defined independently of programming languages, the framework supports the generation of *JCas classes* which are representations of UIMA types as classes in the Java language. In this way, annotations become regular Java objects, features are made accessible via getter and setter methods, and the inheritance mechanism in UIMA type systems is mapped to the Java inheritance mechanism. These classes are pre-generated either manually, after changes in the type system definition have been made, or at best automatically at compile time. If the additional definitions of our *Token* type are not known at generation time, the getter and setter methods for the additional features will not be present in the JCas classes. We may even end up with multiple incompatible versions of the same class on the classpath. JCas classes are very convenient to program with and are commonly used in analysis components. This may be the reason why type merging is not used by any of the component collections.

[2] This decision is somewhat arbitrary, but might be justified as treating the *forward* direction as going from the concrete to the abstract, e.g. from a concrete token to its abstract lemma, or to a more abstract phrase.

Discussion

There are several benefits and drawbacks which should be considered when choosing a strategy for associating feature structures with each other. The strategy should be chosen based on the known requirements:

1. **Does the association itself bear features?** – If the association bears features, it should be modeled as a feature structure.[3] Such features can be generic information like *id* or *confidence*, or domain-specific features, such as the *role* of an association (cf. *RelationArgument* in Figure 5.5 on page 109).

2. **Is it known at type system design time that the association exists?** – If the type system designer knows of the association, it can be modeled explicitly, but if it is not known, the designer cannot add the necessary features, and analysis components cannot fill these features.

3. **Can the association be multi-valued in some cases, although it is normally single-valued?** – If normally a single lemma is associated to a token, then it would be inconvenient to turn the feature which associates a token to its lemma into a multi-valued feature, because extra code would be required while navigating from token to lemma. However, in some cases, a user may run multiple lemmatizers, e.g. to compare their results. In this case, multiple lemmata are associated to the same token via co-indexing, but the token explicitly refers only to one of them, e.g. the most frequent one.

4. **Is the association used very often and is fast navigation between the associated feature structures therefore important?** – If navigation along the association in one or both directions is frequently performed, co-indexing may significantly reduce performance because on every navigation a search through the feature structures must be performed in order to find co-indexed FSes.

5. **Is the association part of an add-on and can therefore not be considered when designing one of the associated feature structure types?** – If the association is unknown when one of the participating types is designed, it is not possible to add a feature to that type referencing the other one. Only backward-linking is possible in this case.

6. **Are the associated types defined in different modules of the collection or framework?** – If the associated types are defined in two different modules of a processing framework or component collection, only a forward or a backward link can be used. A forward/backward link would introduce a circular dependency between the modules.

7. **Are the types involved in the association in the same module?** – Forward/backward linking is only possible if the associated types are defined in the same module.

Table 5.1 summarizes which strategies can be used depending on the requirements.

Co-indexing has the fewest drawbacks. It should be used as the default when performance is not crucial and the association does not bear any features. When the annotation is performed manually, unnecessary effort for the annotator should be avoided. E.g., if annotations are sufficiently associated with each other by co-indexing, the annotator should not be forced to set up additional explicit links between them. If such links are desirable for fast navigation during subsequent automatic processing steps, the links should be set up automatically.

[3] In general, this is not necessary unless the association is *n-to-n*, but it could be considered an unconventional approach to push the features to either of the associated sides, rather than leaving them on the association itself.

Backward linking is the second-best option, although we found that the more common navigation direction tends to be forward, e.g. from accessing a part-of-speech label from a token (forward), instead of accessing a token from its part-of-speech label (backward). Hence, backward linking may not solve performance issues.

Forward linking often addresses navigation performance issues, but should be used with care because it can create potentially undesired dependencies between modules. Consider the type *Token* being defined in a module called *segmentation* and the type *Constituent* being defined in a module called *syntax*. At the lowest level of the syntax tree, a constituent is equivalent to a token. In a forward-linking scenario, the *Token* type would have a feature *parent* pointing to the next higher constituent. However, segmentation is a more basic task than syntactic parsing, and one might not want to make an assumption on constituency structures when defining and implementing segmentation. In a backward-linking scenario, the *Constituent* type has a multi-valued feature *children*, by which it can refer to other constituents or tokens.

Forward/backward linking should be reserved for exceptional cases where the two involved types are closely related to each other, defined in the same module, and fast navigation is important. Note that performance issues with backward linking can be alleviated by creating a temporary reverse index within an analysis component when extensive forward navigation is expected.

Given the friction between type merging and the convenient JCas classes, type merging is not used as a design element in readily available type systems and should probably be avoided when designing new types. However, type merging could become an important design element if these frictions are resolved. One of the UIMA developers recently suggested generating JCas classes just in time, instead of pre-generating them.[4]

Table 5.1: Under which conditions can an association strategy be used?

	Co-indexing	Forward	Backward	F/B
1) Association has features?	-	+	+	+
2) Association unknown?	+	-	-	-
3) Optionally multiple?	+	-	-	-
4) Frequently navigated?	slow	+	+	+
5) Unknown module?	+	-	+	-
6) Known different module?	+	+	+	-
7) Same module?	+	+	+	+

5.1.3.3 Label patterns

A label is a piece of information which could easily be represented as a primitive feature in a feature structure. Consider a type *Token*. The part-of-speech tag for the token could easily be stored as a string value in a feature *pos*. In many cases, type system designers create dedicated types for these labels. For example, in the DKPro Core type system, the *pos* feature is not a string value, but a reference to another annotation of the type *POS* (short for *PartOfSpeech*) which in turn has a feature *value*. This section examines different strategies for modeling labels, as they have been used in the examined type systems.

[4] UIMA developer mailing list post: http://markmail.org/thread/u6bvabdsxiw4agsf (Last accessed: 2013.08.10)

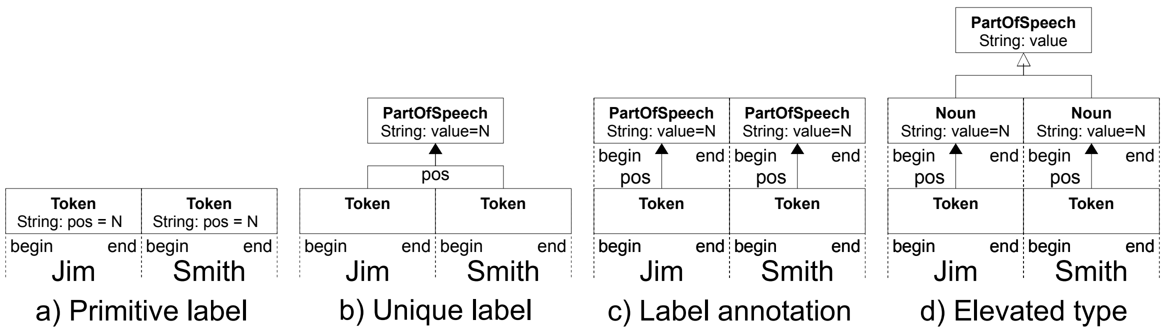


Figure 5.8: Label modeling strategies: part-of-speech tags

Primitive label [Figure 5.8 a)]

A label can be represented as a simple primitive feature on an annotation. Considering that a part-of-speech tag can only be assigned to a token after the token has been identified, the obvious location to place the label feature would be on the *Token* annotation.

For part-of-speech, lemma, and stem information on the *Token* type, this strategy is consistently used only by the ClearTK type system. On other types, however, the use of primitive features for labels is common across all type systems.

Unique label [Figure 5.8 b)]

Instead of maintaining a label as a feature on its associated annotation, a separate type can be created which bears the label feature. The feature on the annotation then changes to a reference to the respective label type. Consider again the *Token* annotation. We add a new annotation type *PartOfSpeech* which has a primitive string feature *value* for the part-of-speech tag. The *pos* feature of the *Token* annotation then references such a *PartOfSpeech* annotation.

U-Compare most prominently introduces the concept of a *unique label*. The part-of-speech label type *POS* is derived from the type *UniqueLabel* which is not anchored to the text. One instance of the *POS* type is created for every part-of-speech tag and then referenced from all tokens which bear that tag. Unique labels are also used for dependency relations and constituents.

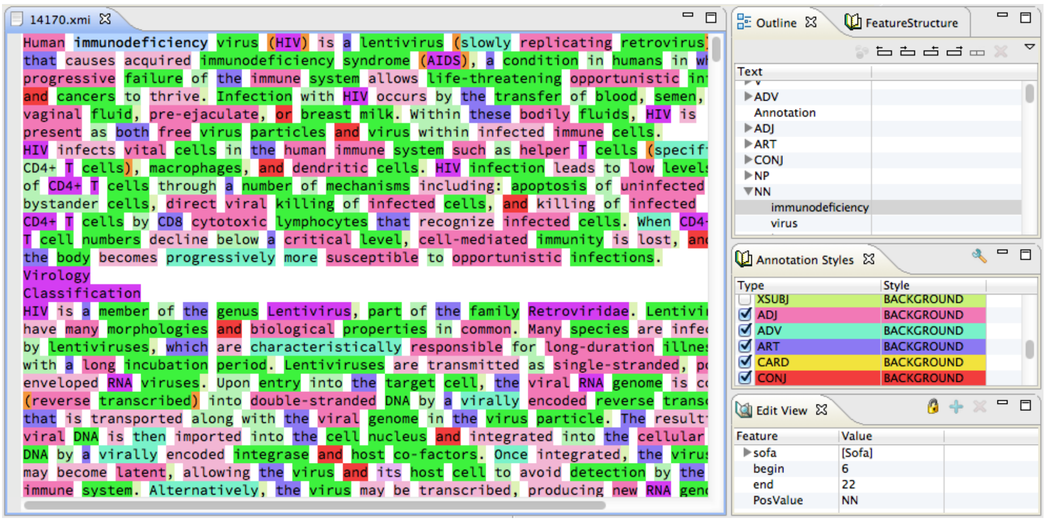
Label annotation [Figure 5.8 c)]

Other type systems derive label types from *Annotation* and anchor them to the text. Consequently, one label annotation exists for each labeled annotation, e.g. one *POS* annotation for each *Token* annotation. The association between the label and the labeled annotation can be expressed through co-indexing or through explicit linking.

Compared to unique labels, this approach incurs a massive duplication of information. However, if the labels have additional information, such as *confidence* or if labels must be addressable via an *id*, the use of label annotations is well justified. Otherwise, such additional information would need to be pushed down to the labeled type, e.g. by adding a feature *posConfidence* to the *Token* type, which would remove to ability to have uniform feature names for the confidence across types.

Elevated types [Figure 5.8 d)]

As mentioned before, label annotations incur a massive duplication of information compared to unique labels, without providing any obvious benefit. However, a benefit may be observed when additional types for specific tags or coarse-grained categories are derived from the label types. Several type systems, including DKPro Core and cTAKES implement this approach. To highlight this benefit, we take a short look at how annotations are programmatically accessed or visualized in an annotation editor.



```
Human immunodeficiency virus (HIV) is a lentivirus (slowly replicating retrovirus)
that causes acquired immunodeficiency syndrome (AIDS), a condition in humans in wh
progressive failure of the immune system allows life-threatening opportunistic in
and cancers to thrive. Infection with HIV occurs by the transfer of blood, semen,
vaginal fluid, pre-ejaculate, or breast milk. Within these bodily fluids, HIV is
present as both free virus particles and virus within infected immune cells.
HIV infects vital cells in the human immune system such as helper T cells (specif
CD4+ T cells), macrophages, and dendritic cells. HIV infection leads to low levels
of CD4+ T cells through a number of mechanisms including: apoptosis of uninfected
bystander cells, direct viral killing of infected cells, and killing of infected
CD4+ T cells by CD8 cytotoxic lymphocytes that recognize infected cells. When CD4
T cell numbers decline below a critical level, cell-mediated immunity is lost, and
the body becomes progressively more susceptible to opportunistic infections.
Virology
Classification
HIV is a member of the genus Lentivirus, part of the family Retroviridae. Lentivi
have many morphologies and biological properties in common. Many species are infe
by lentiviruses, which are characteristically responsible for long-duration illne
with a long incubation period. Lentiviruses are transmitted as single-stranded, po
enveloped RNA viruses. Upon entry into the target cell, the viral RNA genome is co
(reverse transcribed) into double-stranded DNA by a virally encoded reverse trans
that is transported along with the viral genome in the virus particle. The result
viral DNA is then imported into the cell nucleus and integrated into the cellular
DNA by a virally encoded integrase and host co-factors. Once integrated, the virus
may become latent, allowing the virus and its host cell to avoid detection by the
immune system. Alternatively, the virus may be transcribed, producing new RNA gen
```

Text	
▶ADV	
Annotation	
▶ADJ	
▶ART	
▶CONJ	
▶NP	
▼NN	
immunodeficiency	
virus	

Type	Style
☐ XSUBJ	BACKGROUND
✓ ADJ	BACKGROUND
✓ ADV	BACKGROUND
✓ ART	BACKGROUND
✓ CARD	BACKGROUND
✓ CONJ	BACKGROUND

Feature	Value
▶sofa	[Sofa]
begin	6
end	22
PosValue	NN

Figure 5.9: UIMA Annotation Editor displaying part-of-speech tags elevated to types

In the UIMA and uimaFIT APIs, annotations are accessed primarily by their type names. The same is true for UIMA Ruta [133], a language for the rule-based analysis of text, which can be used e.g. for information extraction tasks in UIMA workflows. Accessing a label via a primitive label, unique label, or label annotation requires imposing a constraint on a feature (Listings 5.1, 5.2). By elevating often accessed labels to types, analysis engine code and Ruta scripts can be made more concise and readable. Some analysis tools, such as the UIMA Annotation Editor [13], are hardly usable without elevated types. This is because the editor visually highlights annotations of different types. If all annotations have the same type (e.g. *PartOfSpeech*) they all appear the same (Figure 5.9)[5].

Instead of elevating all labels to types, e.g. all tags in the Penn Treebank tag set, the elevated types can be used to model more coarse grained categories. For example, DKPro Core provides the type *POS* for part-of-speech tags and derived from this several coarse-grained part-of-speech tags, such as *N* (noun), *V* (verb), etc. These largely resemble the *universal part-of-speech tags*

5 Data obtained from the HIV article on the English Wikipedia using and processed with the OpenNLP segmenter and part-of-speech tagger. (Last accessed: 2013-12-11)

Listing 5.1: uimaFIT: condition on label vs. elevated type

```
// Variant 1: Accessing nouns via a label annotation and a feature value constraint
List<PartOfSpeech> nouns = new ArrayList<PartOfSpeech>();
for (PartOfSpeech e : select(jcas, PartOfSpeech.class)) {
  if ("Noun".equals(e.value)) {
    nouns.add(e);
  }
}

// Variant 2: Accessing nouns via an elevated type
List<Noun> nouns = select(jcas, Noun.class);
```

Listing 5.2: Ruta: condition on label vs. elevated type

```
// Variant 1: Accessing nouns via a label annotation and a feature value constraint
PartOfSpeech{FEATURE("value", "N")}

// Variant 2: Accessing nouns via an elevated type
Noun
```

suggested by Petrov et al. [178]. Such coarse-grained tags apply across languages, while the finer-grained tag sets used in most corpora are language specific. The original, language specific tags are also maintained, in the *value* feature of the *POS* type, which is also inherited by all the coarse grained types. In this way, analysis component developers can choose between the easy to use cross-lingual, coarse-grained tags that have been elevated to types and the original, less conveniently accessible tags (cf. Section 5.2).

The U-Compare type system also elevates tags to types, but it does so in combination with unique labels. These are not anchored to the text, therefore the benefit of easy access as shown in (Listings 5.1, 5.2) does not apply. It is unclear what benefit the elevated types provide in this case.

Not all features are suited to be elevated to types. An elevated type is useful if the elevated feature specifies a *kind of*. E.g. a *noun* is a *kind of* part-of-speech tag. However, consider a type *Morphology* with detailed features for morphological information like gender, number, case, etc. Here *male* would not be a *kind of* morphology, if at all, it could be considered a *part of* morphology. Hence, an elevated type is not useful here.

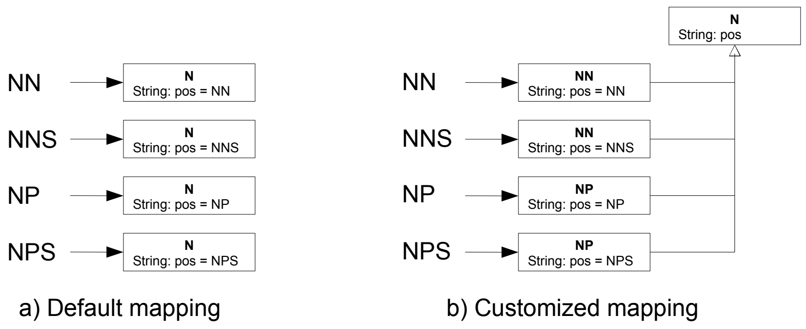


Figure 5.10: Customization of the mapping from tags to elevated types

Elevated types provide an opportunity for users to customize the type system without changing its design (Figure 5.10). Consider a component collection using the universal part-of-speech tags as elevated part-of-speech types. The collection provides a part-of-speech tagging analysis component that generates part-of-speech tags from the Penn Treebank tag set [189], which makes a distinction between proper nouns (pl: *NP*, sg: *NPS*) and common nouns (pl: *NN*, sg: *NNS*). Per default, the analysis component creates an elevated type *N* for all these tags, according to the universal part-of-speech tag set mappings. However, a researcher now requires slightly a more fine-grained mapping, which makes a distinction between the two kinds of nouns, although not between plural and singular. If the mapping from tags to elevated types used by the analysis component is configurable, the researcher can introduce new elevated types *NN* and *NP* and customize the mapping. By deriving the new elevated types from the existing *N* type, this change is even fully compatible with the previous mapping, because any UIMA will automatically find and return the annotations of the type *NN* and *NP* when a search for the more general type *N* is performed. Also, the original tags continue to be preserved in the label feature (*pos*). As a consequence, most analysis components which do not make use of the new, fine-grained distinction will continue to work just as before the change.

Discussion

When a new type bearing a label is designed, it needs to be considered which of the approaches should be used to model the label. If the type system is meant primarily for automatic processing, the primitive label should be the default choice.

The use of unique labels may appear attractive because it avoids the duplication of information. Currently, only U-Compare uses this approach. Unless the set of labels is previously known, maintaining unique labels requires extra overhead, e.g. when a new label is created, an analysis component needs to check if a unique label FS already exists and can be reused or if a new one needs to be created, likewise when a label is changed or removed. If the U-Compare type system is used with analysis components or annotation editors that do not support this bookkeeping, the approach most likely degrades to a non-unique labels. However, editing a document annotated with unique labels in an annotation editor may be problematic, because if the user changes the label associated with one annotation, the change also affects all other annotations associated with that label. Given that unique labels are uncommon, this is likely to be contrary to the user's expectation of a local change affecting only one annotation. Depending on the user interface provided by the annotation editor, the user may not even be able to see a distinction between a unique label and the non-unique label.

Non-unique label annotations, like primitive labels, incur a massive duplication of information, but are easier to handle than unique labels. This is in particular true when additional per-label information, such as an id or a confidence score need to be maintained. They can also serve as extension points for users who wish to add new features or to set up elevated types derived from the label annotation type. However, extra care also needs to be taken to remove the label annotations when the labeled annotations are removed. E.g. stopwords may be removed from text simply by deleting the associated *Token* annotations. In this case, all label annotations (e.g. lemma, stem, part-of-speech tag) should also be deleted.

Most of the examined type systems combine elevated types with label annotations. If analysis components are implemented in such a way that the mapping of tags to elevated types is customizable, this approach offers great flexibility to the user. The output of analysis components can be customized without changing the principle type system design, such that the change can remain oblique to any analysis components that are unaware of it. These components can continue to work as before.

Some annotation tools, such as the UIMA Annotation Editor [13], are unable to visualize annotations differently based on feature values, but only based on their type. Thus, elevated types have the additional benefit of being visually distinguishable in such tools. However, we consider the ability to visualize annotations differently based on their feature values primarily a requirement for the tools. It should not strongly influence the type system design.

Finally, elevated types can help to abbreviate the code for analysis engines, because UIMA and uimaFIT primarily access annotations by their types. However, we see some room for improvement of the APIs and the scripting language to allow more concise code even in the absence of elevated types. E.g. uimaFIT could provide a fluent-style API [89] to apply additional filter predicates on its annotation selection methods. We already discussed such an approach in the uimaFIT development team.[6]

At the present time, we consider elevated types a good choice for commonly used labels, such as part-of-speech tags, syntactic categories, or dependency relations. The label values should be a closed set, of preferably coarse grained categories that can apply for multiple domains or languages.

5.1.3.4 Layer patterns

Layers are a way of organizing annotations. They are very similar to the annotation sets, but have a different focus. In a set, the members of the set share common features, which are moved to the set annotation instead of being repeated for every set member. A layer, in turn, is used to group annotations from the same source (e.g. produced by a particular analysis component) or related to the same linguistic layer (e.g. part-of-speech tagging, constituency

6 `https://code.google.com/p/uimafit/issues/detail?id=65` (Last accessed: 2013-12-11)

structure, dependency structure, etc.). However, an annotation layer is usually not equivalent to a linguistic layer. For example, the linguistic layer *syntax* can be encoded in multiple annotation layers, e.g *constituency structure* and *dependency structure*.

Background

Historically, terms like multi-layer annotation, multi-level annotation, or multi-tier annotation came up because it was not normal for analysis tools to support richly annotating corpora. It was not easily possible to represent certain linguistic annotations with the document-oriented XML, popular for being human readable, as well as machine readable (cf. [70]). In particular overlapping segments, crossing edges, and non-projective tree structures have been a concern. In this context, the introduction of layers was a technical solution to overcome limitations in the meta model used to represent linguistic annotations in XML and to query them (cf. [215]). The idea evolved into treating layers not as containers for annotations at different linguistic levels, but rather as general containers for annotations. Related to the linguistic search engine ANNIS, Zeldes et al. [233] specifically contrast this generic definition of layers to the linguistically motivated layer idea:

> *By multi-layer we mean that the same primary datum may be annotated independently with (i) annotations of different types (spans, DAGs with labeled edges and arbitrary pointing relations between terminals or non-terminals), and (ii) annotation structures that possibly overlap and/or conflict hierarchically. While the term multi-layer itself only implies several types of annotation, such as part-of-speech tagging or lemmatization [...], we use this term to refer more specifically to annotations that may be created independently of each other, annotating the same phenomenon from different points of view, or different phenomena altogether.*

Explicit layers

The U-Compare framework apparently builds upon the idea of layers as generic containers for annotations. The framework allows comparing the output produced by different analysis components of the same type, e.g. different part-of-speech taggers. It relies on a special facility to group and isolate the data produced by each component from each other within the CAS. This is implemented as a set of annotation types (e.g. *AnnotationGroup*) combined with logic which allow managing these groups, e.g. extract and copy them to another CAS (cf. [129]). This is very similar to the concept of annotation layers. It is also a case where the layer is made explicit using dedicated annotation types and FSes, which explicitly reference all the annotations belonging to the layer.

This concept of modeling layers explicitly within the CAS using dedicated annotation types is unique to U-Compare. The other examined type systems do not provide similar mechanisms.

Implicit layers

The other examined annotation type systems do not offer explicit layers. Returning to the linguistic motivation of layers, however, the types themselves can be seen as layers. E.g. the *Constituent* type used to model parse trees in the DKPro Core type system could be seen as defining a *constituency* layer.

One could define a layer as the set of annotations of a particular type and its subtypes. It is possible, however, for the types related to a layer not to have a common supertype. For example, in the DKPro Core type system, the types *CoreferenceChain* and *CoreferenceLink* would make up an implicit coreference layer, but their common supertype is a generic base type for annotations. All other annotations also are derived from this type, so it is not a suitable basis for defining a layer. However, both types are defined in the same namespace (*de.tudarmstadt.ukp.dkpro.core.api.coref.type*), which could be another information used to define a layer implicitly.

But is it useful at all to define layers implicitly in terms of interacting annotation types? In the context of analysis tools, the visualization of analysis results, or search, it can be convenient for the user to operate in terms of these implicit layers. Consider a richly annotated document. Displaying it in an annotation editor can easily overload the visual appearance of the user interface (cf. [232]). It helps if the user can selectively hide or show certain *layers* of annotation, e.g. the constituency structure or coreference chains without having to individually hide each annotation type belonging to that layer

Discussion

Layers are a method of introducing an additional means of organizing the annotation data. UIMA has a similar, but not equal concept called *views*. The CAS is not limited to one primary data object, but can accommodate multiple parallel primary data objects, one per *view*, each with their own FSes. It is possible for an FS in one view to reference an FS in another view. Layers are orthogonal to views, as a single view can contain multiple explicit or implicit layers and these layers can include annotations in multiple views.

Views are a well-supported mechanism in UIMA, in the sense that analysis components can either be aware or not aware of their existence and both cases can be handled conveniently by the UIMA framework. When the analysis workflow is assembled, the component unaware of views can be mapped to a particular view on which it should operate. Components that are aware of views can access as necessary, e.g. to perform a comparative analysis.

Explicit layers are a unique feature of the U-Compare framework. Fortunately, U-Compare is implemented in such a way, that individual analysis components do not need to be aware of the layers, as these are handled at a different level, e.g. by the workflow controller.

Implicit layers are a matter of convention and, unlike views and explicit layers, they do not require specific support from the processing framework or analysis components. We consider the treatment of the namespace as a hierarchy of implicit layers as the best approach. Tools that visualize annotations can easily provide a hierarchical type navigator to allow showing or hiding all types in a namespace.

To summarize, we can say that implicit, as well as explicit layers are useful concepts. Implicit layers are easier to handle if all annotation types belonging to a conceptual layer reside within a common namespace. However, this entails that the namespace structure must be carefully planned when designing a type system. Unless layers are an integral part of the processing framework being employed, most analysis components should not be made aware of layers. An exception may be components which perform comparison operations between layers, e.g. to calculate inter-annotator agreement in cases where each explicit layer represents the annotations from one annotator. The use of views should be considered as an alternative to explicit layers, because they are directly supported by the UIMA framework.

5.1.4 Analysis

Initially, we aimed to provide guidelines and best practices for the design of type systems, to evaluate the potential for a common type system that can be used by multiple component collections, and to determine if type systems are sufficiently expressive to be used in manual annotation tasks. Along with the design pattern analysis, we have provided guidelines supporting type system designer in their design decisions (Section 5.1.3). In this section, we address the remaining two points: the potential for a common type system and whether the type system is sufficiently expressive.

5.1.4.1 Towards a common type system

Even though the examined type systems are all meant for linguistic annotations at levels such as segmentation, part-of-speech tagging, constituency parsing, etc., there are considerable differences. It would not be sufficient to simply rename types and features in order to convert analysis results from one type system to another, e.g. in order to use components from different collections in the same workflow.

To illustrate the differences between the type systems, let us take a look at two very simple types appearing in all of them: *Token* and *Sentence*. Although it may appear that these are trivial types, we will find that there are sufficient differences between the type systems, to make them fundamentally incompatible, even on this basic level.

Sentence

The annotation of sentences is one of the most basic and simple annotations. However, we can already note differences between the type system here (Figure 5.2). In DKPro Core, the sentence does not bear any special features. As any other annotation, it bears a start and end position.

The situation is similar in ClearTK, U-Compare, and JCoRe. However, these type systems base most of their annotations on generic types, such as *ScoredAnnotation*, *J-Annotation*[7], and *BaseAnnotation*, which mainly allow recording confidence information. In ClearTK, the confidence score is stored in a numeric feature *score* (type: double), while in JCoRe, it is stored in a feature called *confidence* (type: String). In U-Compare, the confidence is accessible as *confidence* (type: float) via the complex feature *metadata*.

On top of this, there are even more features. E.g. in U-Compare, many annotations – including *Sentence* – inherit from the *DiscontinousAnnotation* which allows the annotation to have fragments of arbitrary annotation types again. In JCoRe, the additional features *id* and *componentId* are present, which can be used to indicate which analysis component has created the sentence and to assign an identifier to the sentence.

These additional features range from rather questionable to useful. E.g. it can be useful to maintain an identifier information when annotations are ingested from a corpus format which contains such identifiers. It may be reasonable to maintain a confidence score on annotations, but it raises the question if that entails that an analysis component should actually produce multiple annotations with different confidence scores and how other components in the workflow should react to this - in particular when the respective annotation is as basic as a sentence. We have not yet come across any use-case that would require discontinuous sentence annotations. Even if a sentence was interrupted by some other element, e.g. a footnote, there should be alternative ways of handling this without forcing the analysis components to specifically handle discontinuous sentences (cf. [71]).

If the analysis components in the respective component collections actually make use of all these features, transforming data between the type systems requires complex structural operations, and in some cases it incurs a loss of information.

Token

The type *Token*, like *Sentence*, is very basic. In its simplest form, it would also just be marking a span of text without any additional features. However, none of the examined type systems offer this most basic concept of a token. The simplest variant is provided by ClearTK. Here, the token has three primitive string label features for the part-of-speech tag (*pos*), the *stem*, and the *lemma*. However, this simplicity is offset by the fact that *Token* inherits from *ScoredAnnotation*,

[7] This type is actually called *Annotation* (*de.julielab.jules.types.Annotation*), but to make it obvious that it is not the same as the built-in UIMA *Annotation* type (*uima.tcas.Annotation*), we call it *J-Annotation* here.

Table 5.2: Comparison of the *Sentence* annotation type

	DKPro Core 1.5.0	ClearTK 2.0.0-SNAPSHOT	U-Compare 2.2	JCoRe 2.6.8	cTAKES 3.0.0
Supertypes	Annotation	Annotation ScoredAnnotation	Annotation DiscontinuousAnnotation BaseAnnotation SyntacticAnnotation	Annotation J-Annotation	Annotation
Type	Sentence	Sentence	Sentence	Sentence	Sentence
Features	- -	- -	- -	- -	sentenceNumber segmentId
Features inherited from supertypes					
SyntacticAnnotation	-	-	-	-	-
BaseAnnotation	-	-	metadata ->	-	-
DiscontinuousAnnotation	-	-	fragments[]	-	-
ScoredAnnotation	-	score	-	-	-
J-Annotation	- - -	- - -	- - -	confidence componentId id	- - -

Annotation	begin
	end
		begin
		end
			begin
			end
				begin
				end
					begin
					end

which adds the confidence feature *score* via inheritance. The DKPro Core type system also offers part-of-speech tag (*posTag*), *stem*, and *lemma* features, but consistently uses *label annotations* instead of primitive labels. Some type systems even allow assigning multiple labels of the same kind to a token. For example, cTAKES allows a token to bear multiple lemmata, while JCoRe allows a token to bear multiple part-of-speech tags. In U-Compare, there are even multiple possibilities to store the part-of-speech tag. One is the *posString* feature, another is using a unique label referred via the *pos* feature. There are even multiple kinds of unique labels that can be used: the *UnknownPOS* and a whole hierarchy of *elevated label* types.

Summary

Given that there are already great differences between the different type systems or even the most basic of annotation types, we shall not examine further types in detail at this point. As we have seen, the type systems range from being relatively simple, using a relatively flat inheritance hierarchy and few features (e.g. DKPro Core and ClearTK), to quite complex, using a deep inheritance hierarchy introducing many features (e.g. U-Compare and JCoRe). What can we say about the possibility of creating a common type system for all the component collections or about making analysis components configurable in order to work with different type systems?

An analysis component is often a wrapper around an existing analysis tool, functioning as an adapter between the type system used by the analysis workflow and the data structures used by the analysis tool. It is possible, to implement this wrapper in such a way that the names of types and features are configurable. This is only feasible if structural assumptions can be made about the type system, e.g. that the type system uses exclusively primitive labels. However, looking at the type systems examined here, we note that such an assumption would not be valid for most of them. If the type systems are more complex, such as allowing for multiple part-of-speech tags or using elevated types, making analysis components configurable to handle such type systems is not trivial. The primary functionality of the wrapper is the adaption between the wrapped analysis tool and the type system, and extracting this would mean that major portions of the wrapper code into configurable strategies (cf. Section 3.2). Almost no functionality would remain in the wrapper itself. We wrapped numerous analysis tools for the DKPro Core collection (Section 5.2) and found that it is in most cases simpler to implement a new wrapper

Table 5.3: Comparison of the *Token* annotation type

	DKPro Core 1.5.0	ClearTK 2.0.0-SNAPSHOT	U-Compare 2.2	JCoRe 2.6.8	cTAKES 3.0.0
Supertypes	Annotation	Annotation ScoredAnnotation	Annotation DiscontinuousAnnotation BaseAnnotation SyntacticAnnotation Token POSToken	Annotation J-Annotation	Annotation
Type	Token	Token	RichToken	Token	BaseToken
Features	posTag.posValue stem.value lemma.value - -	pos stem lemma - -	posString (POSToken) - base - -	posTag[].value stemmedForm lemma - -	partOfSpeech - lemmaEntries[].key normalizedForm tokenNumber
Subtypes	JapaneseToken	-	-	-	ContractionToken NewlineToken NumToken PunctuationToken SymbolToken WordToken

Features inherited from supertypes

POSToken	-	-	pos->	-	-
SyntacticAnnotation	-	-	-	-	-
BaseAnnotation	-	-	metadata ->	-	-
DiscontinuousAnnotation	-	-	fragments[]	-	-
ScoredAnnotation	-	score	-	-	-
J-Annotation	- - -	- - -	- - -	confidence componentId id	- - -

Annotation	begin
	end
		begin
		end
			begin
			end
				begin
				end
					begin
					end

when a different type system is used than to make a wrapper parametrizable with respect to the type system.

From our analysis, we see a clear potential for DKPro Core and ClearTK to adopt common types, such as a common representation for tokens, sentences. We also see potential for a common type system to represent syntactic parse trees, although these were not discussed here. At the level of *Token* and *Sentence*, a few points remain that need to be discussed:

- Is a confidence score necessary at the token and sentence level? (the ClearTK feature *score* is not present in DKPro Core)
- Should primitive labels (ClearTK) be used, or label annotations (DKPro Core *stem* and *lemma* features) and elevated types (DKPro Core *posTag* feature)?

Types used for further concepts, such as dependency relations, coreference, or semantic role labelling require further analysis and discussion, as the level of detail between ClearTK and DKPro Core is very different.

5.1.4.2 Manual analysis

The information contained in the definition of type systems does not fully describe how analysis components or annotation editors are allowed to interact with the type system. The type system is part of the API of an analysis component. The component makes certain assumptions about how the types from the type system are used.

Consider, for example, dependency relations. These are relations between two tokens, a *dependent* and a *governor*. Consequently, in the examined type systems, dependencies are modeled as some kind of relation type between annotations of the type *Token*. However, there is usually an additional rule not explicated by the type system: there cannot be two dependency relations with the same dependent. For example, the CoNLL-X shared task on multi-lingual dependency parsing [35] used a tab-separated file format (Listing 5.3) to represent dependency annotated sentences in which each dependent token (second column, *FORM*) could have one governor (seventh column, *HEAD*). An analysis component writing results to a file in this format will assume that this rule is observed. Similarly, a component rendering the dependency structure as a tree will assume this rule is observed. This means, the code of such components will be written in such way that it produces bad output or crashes if this rule is not observed.

Listing 5.3: Sentence with dependency annotations represented in the CONLL-X format

```
ID      FORM    LEMMA   CPOSTAG POSTAG  FEATS   HEAD    DEPREL  PHEAD   PDEPREL
1       The     the     DT      DT      _       4       det     _       _
2       quick   quick   JJ      JJ      _       4       amod    _       _
3       brown   brown   JJ      JJ      _       4       amod    _       _
4       fox     fox     NN      NN      _       5       nsubj   _       _
5       jumps   jump    VBZ     VBZ     _       _       _       _       _
6       over    over    IN      IN      _       5       prep    _       _
7       the     the     DT      DT      _       9       det     _       _
8       lazy    lazy    JJ      JJ      _       9       amod    _       _
9       dog     dog     NN      NN      _       6       pobj    _       _
10      .       .       .       .       _       _       _       _       _
```

Since, in practice, most, if not all, analysis components producing dependency relation annotations observe this rule, users working with these components and analysis components consuming these annotations can well rely on it. However, if annotations are manually created with an annotation editor, a user could easily, even unintentionally, violate this rule. At least at the level of UIMA type systems, there is currently no way to specify such additional constraints in a machine readable manner. Unfortunately, such constraints are often not even documented for human consumption in the documentation of the respective type systems.

Apart from UIMA type systems, other technologies, such as the combination of XML [225], XML Schema [226], and Schematron [120], allow the definition of complex constraints which could cover such rules and which XML-based analysis tools could employ to ensure that human annotators do not create any technically possible annotations, but also observe higher-level rules and constraints regarding the interaction of different annotations. GATE permits the definition of annotation schemes using XML Schema, which is comparable to the UIMA type systems, but does not support higher-level rules which would require Schematron. Future work should examine if these technologies can be adapted to non-XML meta models, such as the UIMA CAS.

For the WebAnno project, we plan to permit the user adding custom annotation types (or rather layers, see below). When a custom type is created, a structural design needs to be chosen, such as *span*, *relation*, or *chain*. For each of these, we plan to provide constraints, such as *span cannot cross sentence boundary*, *span boundaries must correspond to token boundaries*, or *relation endpoint must be unique*. Given that it is currently not possible to express such constraints in terms of the type system, and given that there is currently no alternative constraint checking technology which is compatible with the UIMA CAS, we plan to hardcode these constraints. However, the inventory of constrains that we plan to implement in WebAnno can later serve as a point of reference to define requirements towards a generic system for defining and checking constraints.

5.1.5 Summary

In this section, we have examined type systems from five different analysis components for the UIMA processing framework. We have identified several patterns in the design of these type systems, have discussed their benefits and drawbacks, and have provided guidelines when to apply them. In particular, we have discussed implications of individual patterns regarding extensibility and customizability.

Additionally, we have compared the type systems directly to each other by examining the representation of tokens and sentences in detail. We have come to the conclusion that there are significant differences in the design of most of the type systems, which is a major problem when aiming at the design of a common type system used by multiple component collections. The DKPro Core and ClearTK type systems currently appear to be the best candidates to start discussing about a common type system, because compared with the others their type systems are the most similar starting at the lower level annotations, e.g. for sentences and tokens.

We hope that our analysis provides valuable information for type system designers and analysis tool developers. For example, our analysis provides developers of annotation editors with a set of patterns to expect in type systems. They can then offer user interfaces tailored specifically to these common patterns and offer a better interoperability with type systems based on these designs.

In the future, we plan to further examine the feasibility of consolidating different annotation types systems, in particular those of the different UIMA-based analysis component collections. The present work can serve as a basis for informed discussions with the providers of these type systems.

5.2 Component collection

In this section, we present a revised and extended edition of our *DKPro Core* component collection. It uses our techniques to improve usability (cf. Chapter 3) and provides researchers with controllable, portable components as the basis for reproducible research (cf. Chapter 4). We compare DKPro Core to other component collections and discuss the kinds of component collections and their underlying motivations. Finally, we analyze to what degree analysis components are interchangeable in the workflow, their conceptual interoperability, and we make a note of the aspects of provenance and attribution.

Processing frameworks pave the way for interoperable and interchangeable analysis components. However, most state-of-the-art language analysis tools are provided by their authors as standalone analysis tools. The *DKPro Core* collection of analysis components integrates many of these state-of-the-art tools with the Apache UIMA [10] framework. By integrating multiple alternative tools for the same analysis tasks, e.g. different implementations of part-of-speech taggers or parsers, it provides a rich choice and the possibility to select the tools best suited for the task at hand. The DKPro Core collection provides a better coverage of different languages or domains than the single tools alone, which tend to focus on particular languages or domains.

In our overall scenario, this addresses the following issues (Figure 5.11):

❶ **No comprehensive set of interoperable automatic analysis components is available.** DKPro Core provides *analysis assemblers* with a comprehensive broad-coverage collection of interoperable analysis components. *Analysis developers* benefit from the DKPro Core API when building new interoperable components and can rely on convenient concepts, such as the mechanism for resource loading. *Analysis deployers* benefit from the focus on portability in the DKPro Core collection, as it does not rely on web services for processing.

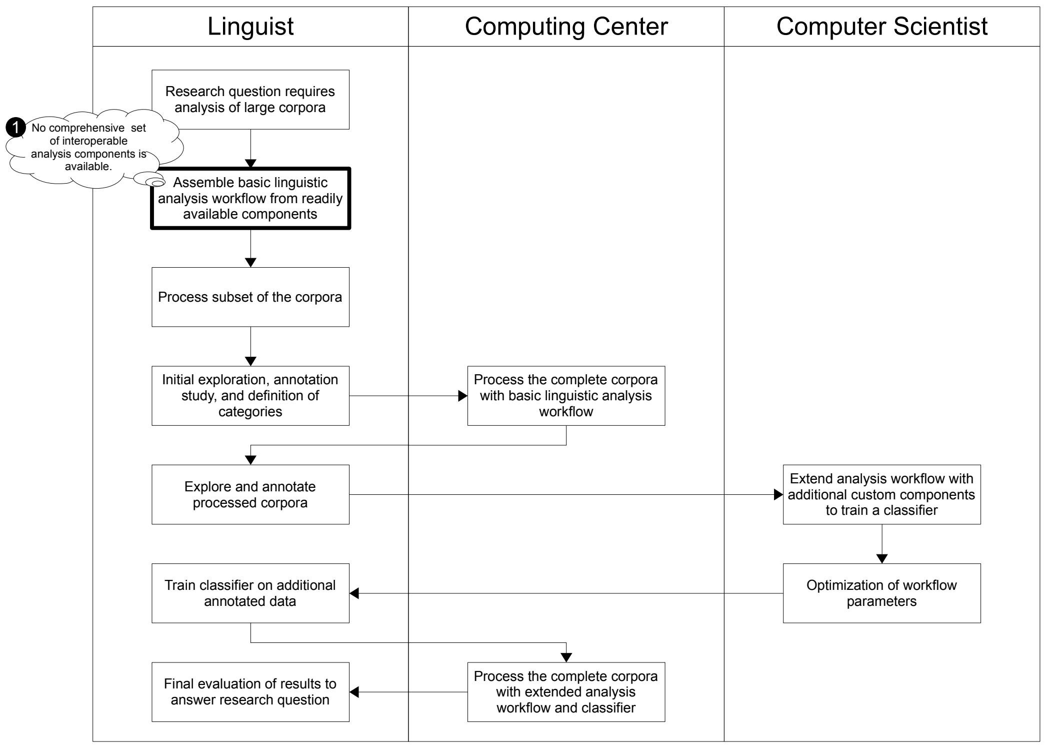

Linguist
Computing Center
Computer Scientist
Research question requires analysis of large corpora
1 No comprehensive set of interoperable analysis components is available.
Assemble basic linguistic analysis workflow from readily available components
Process subset of the corpora
Initial exploration, annotation study, and definition of categories
Process the complete corpora with basic linguistic analysis workflow
Explore and annotate processed corpora
Extend analysis workflow with additional custom components to train a classifier
Train classifier on additional annotated data
Optimization of workflow parameters
Final evaluation of results to answer research question
Process the complete corpora with extended analysis workflow and classifier

Figure 5.11: DKPro Core is a comprehensive broad-coverage collection of interoperable and portable analysis component.

5.2.1 Motivation

When thinking about a collection of analysis components, we primarily expect two requirements to be fulfilled. It should provide us with a *choice* between different components and models, and it should be more than the mere sum of its parts by providing an increased *coverage* of different languages and domains, new components stepping in where others fall short. To reach this goal, we expect the components in a collection to be *interoperable* and *interchangeable*.

Choice

Minimally, an analysis component collection would provide a small set of tools, wrapped for interoperability. However, what we would expect from any project claiming to provide a *real* collection, is choice. Choice among different approaches and algorithms, some of which may be more suited to a task at hand than others. As a simple example, a named entity recognizer based on statistical methods can often be supported by an additional dictionary-based approach. The dictionary-based approach tends to have a high quality (unless the entities are ambiguous), while the statistical approach can help to tackle out-of-dictionary cases. Other reasons to value a choice may be speed, memory requirements, quality, or the license of a particular analysis component or its resources.

Coverage

The ability to replace one component with another one in an analysis workflow enables the user to easily tune the workflow to another domain or language. E.g. a parser with a model trained for the English language can be substituted for another one which comes with a model for German, or one trained on newspaper data can be replaced by another one trained on data from the biomedical domain.

Interoperability

Processing frameworks come with the promise of enabling the interoperability between different analysis tools, but they do not actually provide this interoperability. The interoperability is provided by a common conceptualization of the data exchanged between the analysis components, realized as an annotation type system, as well as wrappers around the tools which adapt the data formats used by the individual tools to the formats used by the processing framework. It is the purpose of a component collection to provide such wrappers. Given the lack of a widely used common annotation type system (cf. Section 5.1), it is currently also the purpose of a component collection to provide an annotation type system .

Interchangeability

While interoperability already entails that components should be interchangeable to a certain degree, there are additional expectations, which warrant treating interchangeability as a separate topic. We consider an analysis component to be interchangeable with another one if it consumes the same input and produces comparable and compatible output. It should also accept largely the same set of parameters and generally exhibit the same behavior. From the perspective of the user, it is desirable that the change of a component is a local operation. I.e. the change should not require further, potentially extensive changes throughout the whole analysis workflow. It should also be a minimal change. This requires a high degree of homogeneity between the components.

5.2.2 State of the art

In this section, we examine several analysis component collections and group them into three general categories: *single vendor collections*, *special purpose collections*, and *broad-coverage collections*. We also briefly mention the integration of machine learning frameworks with processing frameworks.

While interoperability is an inherent trait in all component collections, we find that not all component collections meet our requirements regarding choice, coverage, interchangeability. However, this is not due to the collections being of bad quality, but rather because they have been built with a different objective. We find that our requirements towards choice, coverage, and interchangeability are best met by what we describe as *broad-coverage collections*.

5.2.2.1 Single vendor collections

A single vendor collection provides a set of analysis components for different analysis tasks. For each task, typically only a single analysis component is provided. In order to make the collection useful for a larger community, support for multiple input and output formats is a convenient extra. However, if the collection is sufficiently comprehensive, support for multiple formats may even be unnecessary, because users can comfortably work fully within the ecosystem of the collection. This may cause a *locked-in* situation, in which it becomes difficult for a user to switch to a different collection, because data cannot be easily used with another collection.

Such a collection does typically not rely on an independent processing framework for interoperability, but may include a proprietary interoperability layer for use within the collection. Other types of collections, e.g. special purpose collections or broad-coverage collections, tend to integrate select components or all components from single vendor collections and make them interoperable with components form other vendors.

There are many single vendor collections, such as Apache OpenNLP [9], ClearNLP [48], FreeLing [91], GENIATagger [100], Mate-Tools [148], LanguageTool [135], LingPipe [140] and Stanford CoreNLP [200]. For the sake of brevity, we chose two of them and describe them in more detail.

Stanford CoreNLP

Stanford CoreNLP [200] is a collection of most of the analysis tools provided by the Stanford Natural Language Processing Group,[8] which previously had only been available as separate tools. These include in particular a part-of-speech tagger [219], parser [131; 132; 197], named entity recognizer [86], a coreference resolution system [137], and a sentiment analysis system [198], among other auxiliary tools. The Stanford tools are open-source, popular, and enjoy an excellent reputation.

CoreNLP does not rely on any existing processing framework like GATE or UIMA, but provides its own, lightweight processing framework. CoreNLP does not support all the languages which are supported by the individual Stanford tools. It focusses on English. However, in many cases, it is possible to obtain models for other languages from the distribution of the individual tools and use them with their respective counterparts within CoreNLP. Users can implement custom components for CoreNLP. On the other hand, CoreNLP has been integrated into several analysis component collections, e.g. ClearTK and DKPro Core.

[8] http://nlp.stanford.edu (Last accessed: 2013-10-16)

Apache OpenNLP

Apache OpenNLP [9] is another collection of analysis tools. It covers similar analysis types as CoreNLP but, its analysis is arguably less sophisticated. Yet, publicly available models for OpenNLP cover more and largely different languages than CoreNLP, e.g. Danish, Dutch, or Portuguese. However, not for every type of analysis are there models available for all these languages. English remains the best supported language in OpenNLP.

OpenNLP itself does not provide any kind of processing framework to build analysis workflows using its tools. However, the collection provides a set of wrappers for the UIMA framework. These wrappers are meant to be configurable to work with different type systems. Unfortunately, they expect a specific type system design relying on primitive labels, which makes them incompatible with type systems relying of label annotations (cf. Section 5.1.3.3), such as DKPro Core.

5.2.2.2 Special purpose collections

A collection of components may be compiled for a special purpose. The components do not necessarily come from the same vendor, but they are carefully chosen and tuned for a specific goal. E.g. resources used by the analysis components are targeted at a specific language or domain. It may not be necessary to support multiple different input format or output formats for the analyzed data.

Apache cTAKES

The *clinical Text Analysis and Knowledge Extraction System* (*cTAKES*) [190] provides such a special purpose component collection for the processing of medical records based on Apache UIMA [10]. It consists of a collection of analysis components specifically targeted at the application domain, e.g. annotation of drugs, finding mentions of side effects of drugs, etc. Tools for linguistic preprocessing are collected from different vendors and wrapped by cTAKES analysis components. Typically, only one tool is integrated for each processing task, as the goal of the project is not variety, but domain-specific analysis of high quality. Additionally, domain-specific models are provided for these third-party tools, e.g. part-of-speech tagging models trained on medical data. Components for higher-level analysis are provided by the project itself.

5.2.2.3 Broad-coverage collections

A component collection can be built with the goal of integrating a broad set of tools from different vendors into a common framework. This can increase the coverage, e.g. as different vendors focus on a range of domains or languages. For the convenience of the user, components used for the same analysis task should also be usable in the same way, in particular parameters should have the same names and accept similar settings.

ClearTK

The ClearTK [172] analysis component collection integrates third-party tools for various linguistic analysis tasks based on Apache UIMA [10]. The focus of ClearTK is to provide a component collection, but rather to offer a machine learning toolkit for building new kinds of language analysis (see Section 5.2.2.4). To prepare the data with annotations which can be used by the machine learning toolkit to generate features, ClearTK also includes several analysis components based on different third-party tools, e.g. Apache OpenNLP [9], ClearNLP [48], and Stanford CoreNLP [200]. The selection of these components appears to be based on the project requirements of the ClearTK developers, not on the principled goal to provide a broad choice.

ClearTK additionally includes an original component for the identification of events, times, and temporal relations [22].

JCoRe

The JCoRe collection of UIMA components [109] is relatively small and therefore does not provide much choice. It consists of a set of components centered around the JCoRe type system [38]. Some of these components wrap tools from third parties, such as Apache OpenNLP [9], LingPipe [140], or the MSTParser [163]. Other components are original creations, e.g. a named entity tagger. While most other collections are released and versioned *en bloc*, the JCoRe components are versioned and released individually. There is no full release or overall version for the whole collection.

U-Compare

The U-Compare collection of UIMA components [129] provides many components and considerable choice, but it relies on web services for many of its components. Therefore, it is mostly rather a virtual collection comprised of some portable software components and some services, rather than a collection of portable analysis components. U-Compare wraps tools for part-of-speech tagging, lemmatization, chunking, parsing, named entity recognition, and several other kinds of analysis from various third-party software packages, such as OpenNLP, FreeLing [91], and GENIATagger [100].

Tesla

Tesla [194] aims to integrate a broad-coverage collection and to provide the user with convenient access to these components through a graphical user interface. While other frameworks mainly define the interface between analysis components in terms of the annotation type system, Tesla also considers the behavior of a component and its parameters a part of the interface specification. A component interface specification is called *role* in Tesla and is realized via a Java interface that components implement. Roles can be organized in a hierarchy using Java's inheritance mechanism. Tesla provides a built-in set of such roles which is called the *Tesla Role System* (TRS). In practice, however, it appears that data types are not shared between roles and that every role is tied to one particular data type which may, however, be the root of a type hierarchy (e.g. the type hierarchy for part-of-speech tags). Thus, there are more data types than roles defined in the TRS. However, Tesla integrates only a few tools, mainly those required for the analysis of the allegedly enciphered Voynich manuscripts [112]. So, even though the design of Tesla aims to be generic to integrate many tools, it could be considered a special purpose collection.

GATE

GATE [53], similar to Tesla, also aims to integrate a broad-coverage collection and allows the user to work with it via a graphical user interface. The focus of GATE, however, is not primarily on making existing analysis tools interoperable, but rather to integrate them into the GATE system as a source for automatically generated annotations which then can be used by JAPE scripts, which can be manually corrected by human users, or which can be used as a starting point for human annotators to add higher-level annotations.

5.2.2.4 Machine learning toolkits

While most collections are comprised of ready-to-use analysis components, others focus on offering building blocks for the creation of analysis components. These range from feature extractors to the integration of various machine learning methods from which particular analysis components can be built. Any pre-built components shipping in such a collection may only serve as examples how to build others.

The ClearTK collection has a strong focus on machine learning, providing special support for feature extraction and for different machine learning frameworks in its *cleartk-ml* modules [172]. Based on these, custom analysis components for all kinds of tasks can be implemented, provided that annotated data exists from which the machine learning algorithms can be trained. ClearTK also provides a framework for the evaluation and cross-validation of such custom analysis components.

Toolkits like ClearTK serve as an intermediate layer between analysis components and machine learning libraries, such as Weka [110], Mallet [150], etc.

5.2.3 Contribution: A broad coverage collection of interoperable and interchangeable components

In this section, we present the *DKPro Core* collection of analysis components, which was redesigned and extended as part of the present thesis. It is a broad-coverage collection based on the Apache UIMA framework. In DKPro Core, we integrated many, mainly third-party, analysis tools. To our knowledge, DKPro Core is the component collection integrating the largest number of portable analysis components at the time of writing. This is illustrated by Table 5.4. For the sake of brevity, the table covers only software packages covering multiple analysis steps and omits stand-aloonne analysis tools, such as individual parsers, taggers, etc.

The DKPro Core component collection has already existed prior to the present work (cf. [107]). Yet, the present thesis represents the most detailed treatise and analysis of DKPro Core at the time of writing. It highlights the novel concepts developed within this thesis. The author has acted as a lead developer on the DKPro Core project since 2009. In that function, the collection was extended with many new components (e.g. [71]), and cross-cutting concerns, such as resource loading and the introduction of naming conventions for parameters have been addressed to improve the user experience. Additionally, DKPro Core has been turned from an internal, monolithic project into a highly modular and increasingly popular open-source project. For example, *EXCITEMENT*[9], an open platform for textual entailment, has adopted DKPro Core as part of its preprocessing infrastructure (cf. [166]).

Our experience in building this collection allows us to highlight aspects beyond technical interoperability, that have to our knowledge not been treated sufficiently, e.g. regarding interchangeability and attribution. Beyond that, there is also the problem of conceptual interoperability of tag sets and annotation guidelines, which has already been addressed by researchers working on corpora, but has, to our knowledge, not yet received much attention in the context of processing frameworks and component collections.

5.2.3.1 Goal

DKPro Core is more than just a simple collection of interoperable analysis components. It was built with a focus on improving the productivity of researchers working with automatic language analysis. Our goal is that researchers should be able to focus on their actual research questions, not on the *plumbing* together of heterogenous technologies. The collection aims to attain this goal by following these principles:

- **Choice** – For most analysis steps, we have integrated multiple different tools from different vendors. DKPro Core 1.5.0 covers analysis tasks from coreference resolution, chunking,

[9] *Exploring Customer Interactions through Textual EntailMENT* http://www.excitement-project.eu (Last accessed: 2013-10-16)

Table 5.4: Support for analysis tool across component collections

	DKPro Core 1.5.0	ClearTK 2.0.0-SNAPSHOT	cTAKES 3.0.0	U-Compare 2.2	GATE 7.1	Tesla 1.0.0.201105090947
Apache OpenNLP [9]	yes	yes	yes	yes	yes	-
ClearNLP [48]	yes	yes	yes	-	-	-
FreeLing [91]	-	-	-	yes	-	-
GENIATagger [100]	-	-	-	yes	yes	-
Mate-Tools [148]	yes	-	-	-	-	-
LanguageTool [135]	yes	-	-	-	-	-
LingPipe [140]	-	-	-	-	yes	yes
Stanford CoreNLP [200]	yes	yes	-	-	yes	yes

Note: An entry stating *yes* indicates that at least one of the analysis steps provided by the software package is integrated, not that all steps are integrated. However, in most cases many or all the provided steps are integrated. This table lists only major software packages providing multiple analysis steps (3+). All mentioned component collections additionally support various tools providing only one or two analysis steps. Tools which are integrated via web services are not listed, because this is not a portable integration (cf. Section 4.1). This affects mainly U-Compare which integrates a significant number of proprietary analysis tools via web services.

decompounding, language identification, lemmatization, morphological analysis, named entity recognition, syntactic parsing, dependency parsing, part-of-speech tagging, segmentation, semantic role labeling, spell checking, to stemming. For each task, up to seven different tools have been integrated. Additionally, 19 different data formats are supported.

- **Coverage** – For many of the analysis components, multiple sets of resources for different languages have been packaged and integrated. DKPro Core 1.5.0 integrates 94 models in 15 languages.
- **Interchangeability** – We set up a naming convention for component parameters, and where feasible we take care that they accept the same settings across different components (cf. Section 5.2.3.5). For example, the parameters to manually select a model (cf. Section 3.1.3.1) or to override the mappings for elevated types (cf. Section 5.1.3.3) have the same names regardless of the component. So a user substituting one component for another does not have to learn a completely new set of parameters. At times, even only changing the name of the implementation is sufficient, with the parameters and parameter values remaining the same.
- **Portability** – Analysis components are downloadable and run on different system platforms, either by means of the Java virtual machine, or by providing binaries compiled for different operating systems. DKPro Core integrates with Maven infrastructure in order to provide properly versioned artifacts and deploy these to the user's system. This is an important step towards the creation of portable and reproducible workflows (Section 4.1). The portability is also an important issue because we care for the ability to scale out DKPro Core workflows to a compute cluster in order to process large amounts of data. To maintain maximum control over the processing, NLP web services are explicitly excluded from DKPro Core. Instead, we address the problems of packaging and automatically deploying processing components to the user's computers.
- **Usability** – Analysis components require only minimal mandatory configuration and many components require no mandatory configuration at all because based on the processed data they can automatically determine which resources, e.g. parser or part-of-speech tagger models, are required (cf. Section 3.1). Many of the DKPro Core components are also capable of automatically downloading resources at runtime, depending on the data being processed.

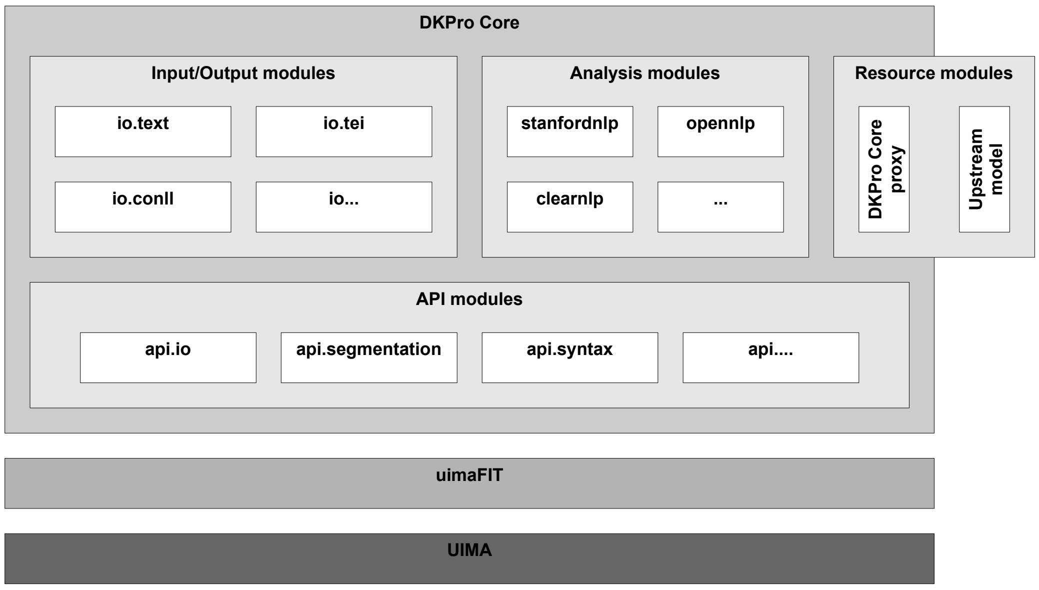


Figure 5.12: DKPro Core architecture

5.2.3.2 Architecture

The DKPro Core component collection consists of four main parts (Figure 5.12): the *API modules*, the *input/output modules*, the *analysis modules*, and the *resource modules*. The collection is fully based on UIMA and uimaFIT.

API modules

The DKPro Core API modules provide the foundation of the collection. They provide the DKPro Core type system which forms the interface used by the components to communicate with each other. Additionally, it provides common functionality which is shared across the components, such as the automatic resource selection functionality (see Section 3.1). This part consists of multiple modules which are typically named after a functional area (e.g. *api.io*, *api.parameter*, ...), or a linguistic layer (e.g. *api.segmentation*, *api.syntax*, ...).

Input/output modules

The DKPro Core I/O modules contain components that allow reading the data to be processed into an analysis workflow and to persist the results of a workflow. Each data format supported by DKPro Core has its own I/O module. The module typically contains two components, one for reading the format and one for writing it. These components build on the DKPro Core API, in particular on the *IO* API module and the annotation types.

Analysis modules

The analysis modules provide wrappers turning mostly third-party analysis tools into UIMA analysis components. Typically, the modules are named after the particular third-party tool or software package that they wrap. Each module provides one or more analysis components, one for each analysis step supported by the wrapped analysis tool. The modules build on the DKPro Core type system and other functionalities provided by the API modules.

Resource modules

Many analysis components require additional resources, such as models for parsers, part-of-speech taggers, etc. DKPro Core employs the split resource packaging mechanism introduced in Section 3.1. Metadata specific to DKPro Core is maintained in a proxy artifact, while the framework-independent resources are packaged in an upstream artifact.

5.2.3.3 Type system

The DKPro Core type system has largely existed in the present form before this thesis. Several additions and enhancements have been made since that time. This section therefore only provides a high-level overview over the type system. Section 5.1 presents an analysis of the type system design and comparison to other type systems.

Figure 5.13 gives a conceptual overview of the type system. All types bear various features, as illustrated for the *DocumentMetaData* type, which otherwise have been omitted for brevity.

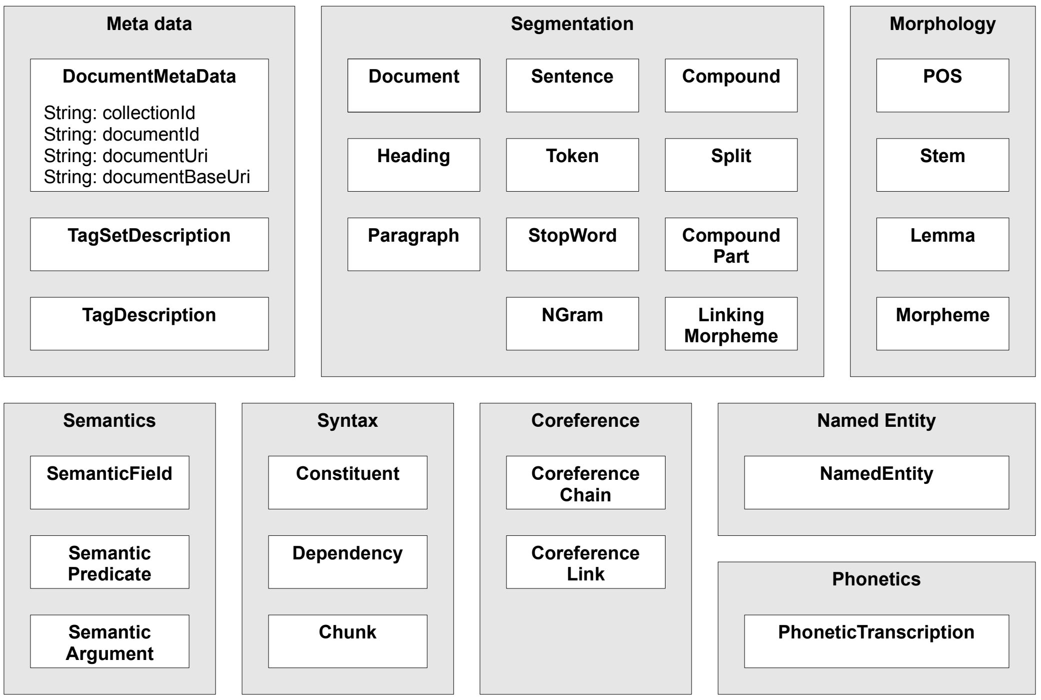


Figure 5.13: DKPro Core type system conceptual overview

The DKPro Core type system is designed predominantly using a combination of label annotations and elevated types (cf. Section 5.1). This yields a relatively flat inheritance hierarchy of three levels for most DKPro Core types (Figure 5.14):

- **Structure** – The top layer is formed by the built-in UIMA type *Annotation* which resembles a segment anchored to the primary data.
- **Generic type** – The second level is formed by *generic label type* (e.g. *POS*, *Constituent* or *Dependency*) which inherits from the UIMA type *annotation* and which bears a *value* feature containing the tag.
- **Coarse-grained category** – The third level consists of a set of coarse grained elevated types (e.g. *Noun*, *Verb*, etc.). While the second level forms a part of the API specification between analysis components, the coarse-grained types of the third level are customizable. Users can their provide own mappings from fine-grained tags to these coarse-grained elevated types, or even derive a completely new set of coarse grained types from the generic second-level annotation type and map to those.

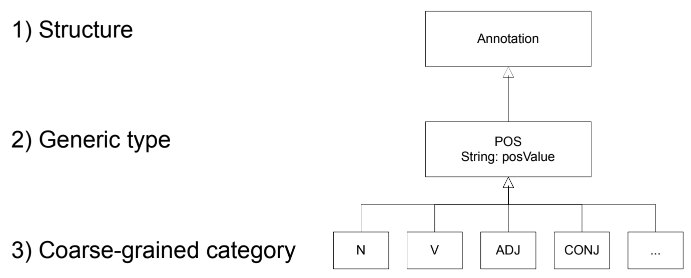


Figure 5.14: DKPro Core type system inheritance hierarchy example for part-of-speech tags

5.2.3.4 Components

Table 5.5 shows the most important analysis tools integrated into the DKPro Core component collection and which analysis steps they cover.[10] The component collection focuses on language analysis steps which are sometimes subsumed under the term *linguistic preprocessing*:

- **Segmentation** – identification of sentence boundaries and tokens.
- **Part-of-speech (POS) tagging** – assignment of part-of-speech tags to tokens.
- **Stemming** – reduction of tokens to a truncated form (stem), typically by removing that portion of a token which can be subject to inflection.
- **Lemmatization** – normalization of a token to a base form from a dictionary, typically the nominative singular for nouns and the infinitive form for verbs.
- **Named entity recognition (NER)** – identification and classification of noun phrases resembling entities such as organizations, persons, locations, etc.
- **Constituency parsing** – identification of the constituency structure of sentences.
- **Dependency parsing** – classification of functional relations between head words within a sentence.
- **Semantic role labelling (SRL)** – identification of semantic predicate-argument structures in sentences.
- **Coreference resolution (Coref.)** – identification of coreferent expressions in a document.

These steps and the analysis tools we integrated for each step have in common that the data they consume and produce is comparatively simple and does not require a very elaborate type system. An area that we currently only start to cover is morphological analysis, which in fact will require annotation types with more fine-grained features, e.g. for gender, number, case, etc. and which may require a more in-depth consideration how to normalize this information and convert it between the different analysis components. The currently integrated analysis steps largely rely on simple tags or string values, which are passed on verbatim between components.

5.2.3.5 Parameters

We consider an analysis component to be interchangeable with another one if it not only consumes the same input and produces comparable and compatible output, but if it additionally accepts largely the same set of parameters and generally exhibits the same behavior. The change of a component should be a local operation and a minimal one that should not require further, potentially extensive changes throughout the whole analysis workflow.

[10] The analysis tools may provide support for additional analysis steps, which have not yet been integrated into DKPro Core. We limited the overview only to the most important analysis steps. There may also be additional analysis steps, e.g. chunking or spell checking, which are covered by the tools integrated into DKPro Core.

Table 5.5: Overview of analysis tools integrated and analysis steps covered by DKPro Core 1.5.0 for the most important analysis steps

Module	Version	Segmentation	POS tagging	Stemming	Lemmatization	NER	Constituency	Dependency	SRL	Coref.
ClearNLP [48]	1.3.1	yes	yes	-	yes	-	-	yes	yes	-
Stanford CoreNLP [200]	3.2.0	yes	yes	-	yes	yes	yes	yes	-	yes
GATE [98]	7.1	-	yes	-	yes	-	-	-	-	-
Apache OpenNLP [9]	1.5.3	yes	yes	-	-	yes	yes	-	-	-
Mate-Tools [148]	3.5	-	yes	-	yes	-	-	yes	-	-
LanguageTool [135]	2.2	yes	-	-	(yes)	-	-	-	-	-
BerkeleyParser [21]	r32	-	yes	-	-	-	yes	-	-	-
MaltParser [144]	1.7.2	-	-	-	-	-	-	yes	-	-
MeCab [155]	0.993	yes	yes	-	-	-	-	-	-	-
Morpha [161]	1.0.4	-	-	yes	(yes)	-	-	-	-	-
MSTParser [163]	0.5.1	-	-	-	-	-	-	yes	-	-
Snowball Stemmer (Lucene) [196]	3.0.3	-	-	yes	-	-	-	-	-	-
TreeTagger [220]	3.2	-	yes	-	yes	-	-	-	-	-

Note: Most of the software packages mentioned above are comprised of more than one analysis tool and possibly multiple different approaches and algorithms per tool, for each of which publications could be cited. DKPro Core does not integrate these algorithms individually, but rather integrates the tools. Instead of citing numerous publications here, we refer to the websites of the individual tools in the bibliography, where the current state of development and often links to publications can be found.

Additionally, repeating common concepts through the API facilitates learning it. For this reason, there are naming conventions in the DKPro API, for example, for configuration parameters. Following the *principle of least surprise* [123; 184], configuration parameters with the same effect should also have the same name on every component (and vice versa). DKPro Core standardizes the names and the base functionality of several commonly used parameters. For some parameters, components may accept additional, non-standard values. It is not required that all components support all the standard parameters. Here is an overview of the most commonly used standard parameters and their effect:

General parameters

- **language** – the language of the documents read if it appears on a reader, or the language of the model to be loaded if it appears on an analysis component. In both cases, the language must be given as a two-letter ISO 639-1 code [122].

Definition: ***principle of least surprise*** – The principle of least surprise is a general principle of design. It simply states that the user should not be surprised, e.g. by inconsistent design or unexpected behavior. The use of common terminology in scientific literature is one application of this principle. Having to press a button with the label *Start* to turn a computer off is a counterexample. (Also known as the *Law of Least Astonishment* in [123]).

Parameters for readers and writers

- **sourceLocation** – the location to read input data from. This can be a path on the file system, an URL, or anything else that the reader supports.[11]
- **sourceEncoding** – the encoding of the input data. This is a very common parameter, because many data formats, in particular corpus formats are text-based.
- **targetLocation** – the location to output the data to. Again, this can be anything the writer supports.
- **targetEncoding** – the encoding used to write the data.

Parameters for analysis component

The following parameters are related to the *Resource Provider API* of DKPro Core (cf. Section 3.1.3.1) and the mapping of tags to elevated types (cf. Section 5.1.3.3):

- **modelLocation** – the location of the model used by the component. At least file system paths, file URLs and classpath pseudo-URLs should be supported. A component may support additional location types.
- **modelEncoding** – the character encoding expected by the model. This is mainly relevant for analysis components that communicate with external processes. For example, TreeTagger [192] models are made for a particular encoding, and this needs to be used when sending data to TreeTagger and reading output from it.
- **modelVariant** – the variant of a model. The variant string provides information on tool-specific parameters, the corpus the model was derived from, or other circumstances under which the model has been created. For example, models for named-entity recognizers might use the model to encode which kinds of named entities a model covers.
- **<layer>MappingLocation** – this is a set of parameters used to override the mapping of tags produced by a low-level analysis algorithms to annotation data types. The placeholder <*layer*> can assume values such as *pos*, *chunker*, *namedEntity*, etc.
- **printTagSet** – when a model was loaded, extract the tag set from the model and display it. This helps the user to verify that the model is really producing the expected tag set.

5.2.3.6 Resources

Many of the analysis components integrated into DKPro Core require additional resources, such as models for parsers, part-of-speech taggers, etc. The general strategy for accessing and packaging resources has already been described in Section 3.1. What remains to be explained here is how the packaging of resources is facilitated in DKPro Core and which metadata specific to DKPro Core is stored in the proxy artifacts.

Resource packaging

The packaging of resources for use with DKPro Core components is performed using a set of macros based on the Apache Ant [6] tool. Typically, a resource is downloaded from its upstream provider and then packaged into a proxy artifact containing the DKPro Core metadata and an upstream artifact containing the original resource. An example of how these macros are used is given in Listing 5.4 (p. 140).

[11] All readers derived from the DKProResourceCollectionReaderBase support file system paths, file: URLs, classpath: URLs, and jar: URLs. Support for additional locations, e.g. smb: URLs for accessing Samba shares or hdfs: URLs for accessing a Hadoop Distributed File System (HDFS) can be plugged in to the ResourceCollectionReaderBase.

Table 5.6: Examples for the *variant* coordinate in DKPro Core

Original file name	Variant	Explanation
en-pos-maxent.bin	maxent	Language and kind of tool omitted
engmalt.poly-1.7.mco	poly	Language, tool name, and version omitted
french.tagger	default	No usable information available
mst-eisner.model.gz	eisner	Kind of tool omitted
wsj-0-18-left3words-distsim.tagger	wsj-0-18-left3words-distsim	Kind of tool is encoded in the extension, full file name is used

Coordinates

As described in Section 3.1.3.1, a set of coordinates is used to uniquely identify the packaged resources: *type*, *language*, *variant*, and *version*.

- **type** – The type typically assumes one of the following values, depending on the kind of tool for which a resource is used:
  - **token** – tokenizer
  - **sentence** – sentence splitter
  - **lemmatizer** – lemmatizer
  - **tagger** – part-of-speech tagger
  - **morphtagger** – morphological analyzer
  - **ner** – named-entity recognizer
  - **parser** – constituency or dependency parser
  - **coref** – coreference resolver
- **language** – The language is represented by a two-letter ISO 639-1 code [122].
- **variant** – The variant depends on the model being packaged (Table 5.6). Typically, this is a part of the upstream model name which allows telling models apart, but does not contain redundant information about the language, kind of tool, or version. If suitable information is available and we know of only one model for the combination of tool and language, we use the variant *default*.
- **version** – For the version, we consistently use the format *YYYYMMDD.XX*, where *YYYYMMDD* is a timestamp and emph *.XX* is used to distinguish different versions of metadata specific to DKPro Core, e.g. information on the tag set. We found not all upstream providers provide version information for their models. Yet, these models change from time to time, as we could observe by calculating a checksum for the upstream files and comparing that with checksums we have observed previously. In the same way, we could also determine that the upstream files sometimes did actually not change, even though there was a change in their version. We update the timestamp based on the timestamp or release date of the upstream version when we observe that the checksum of the upstream file has changed. We update the metadata suffix when we add or update the metadata.

5.2.4 Analysis

In this section, we discuss our experiences and conclusions we draw from building the DKPro Core component collection. To support these, we analyze the components and resources which have been integrated into the collection. We find that most components for the same analysis

Listing 5.4: Packaging a model for the DKPro Core OpenNLP part-of-speech tagger component

```
 1 <!-- FILE: models-1.5/en-pos-maxent.bin - - - - - - - - - - - - - - - - - - - - - - - -
 2   - 2012-06-16 | now        | db2cd70395b9e2e4c6b9957015a10607
 3   -->
 4 <get
 5   src="http://opennlp.sourceforge.net/models-1.5/en-pos-maxent.bin"
 6   dest="target/download/en-pos-maxent.bin"
 7   skipexisting="true"/>
 8 <install-stub-and-upstream-file
 9   file="target/download/en-pos-maxent.bin"
10   md5="db2cd70395b9e2e4c6b9957015a10607"
11   groupId="de.tudarmstadt.ukp.dkpro.core"
12   artifactIdBase="de.tudarmstadt.ukp.dkpro.core.opennlp"
13   upstreamVersion="20120616"
14   metaDataVersion="1"
15   tool="tagger"
16   language="en"
17   variant="maxent"
18   extension="bin" >
19     <metadata>
20       <entry key="pos.tagset" value="ptb"/>
21     </metadata>
22 </install-model-file>
```

- *Lines 4-7: Download resource from upstream provider.*
- *Line 10: Hash sum for the resource.* Some upstream providers do not version their resources. The hash sum is used to detect if a resource has changed. If this is the case, the *upstreamVersion* field needs to be updated.
- *Lines 11-14: Maven coordinates.* The group ID, artifact ID and version of the resulting Maven artifacts.
- *Lines 15-17: Resource coordinates.* Additional resource coordinates, according to the scheme presented in Section 3.1.
- *Line 18: File extension.* File extension used for the packaged resource within the artifact.
- *Line 19-21: Additional metadata.* Metadata specific to DKPro Core. This data is stored within the proxy artifact.
- *Lines 8-22: Package resource.* The command produces two artifacts:
  - *Proxy artifact* – uses the group ID from line 11, the artifact ID constructed as `{artifactIdBase}-model-{tool}-{language}-{variant}`, and the version constructed as `{upstreamVersion}.{metaDataVersion}`.
  - *Upstream artifact* – uses the group ID from line 11, the artifact ID constructed as `{artifactIdBase}-upstream-{tool}-{language}-{variant}`, and the version from line 13. The resource is embedded in this file at the location constructed as `{artifactIdBase}/lib/{tool}-{language}-{variant}.{extension}`.

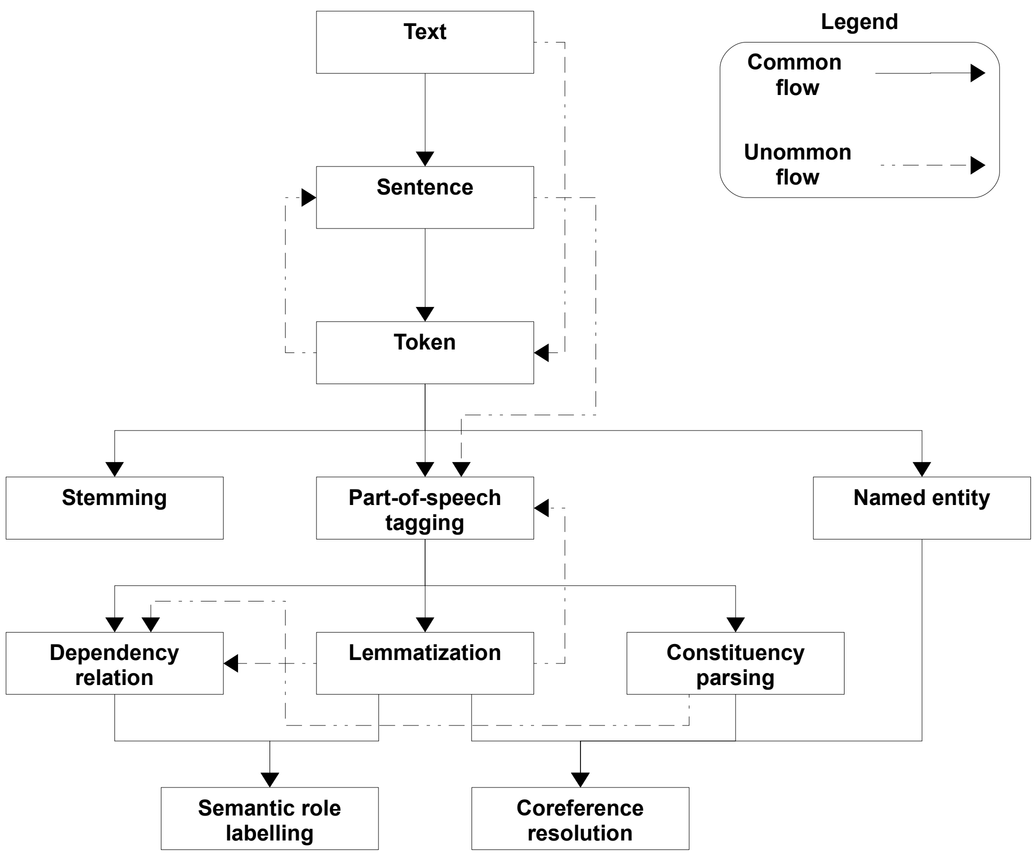


Figure 5.15: Dependencies between analysis steps

tasks take the same inputs, so that exchanging one for another does mostly not entail a restructuring of the analysis workflow. We find that a good interoperability between the components can be achieved when they are used in conjunction with publicly available models, because the models themselves are largely conceptually interoperable, e.g. using the same tag sets. However, we also find that often there is little provenance metadata about the resources, or this data is not easily and readily accessible.

5.2.4.1 Interchangeability

We consider an analysis component to be interchangeable with another one if it consumes the same input and produces comparable and compatible output. It should also accept largely the same set of parameters. We already described a standard set of parameters (Section 5.2.3.5) used across all DKPro Core components to improve their interchangeability. Now, we examine if components addressing the same type of analysis, e.g. part-of-speech tagging or parsing, also accept the same inputs and produce the same outputs. This allows us to determine if the change of a component is a local operation. I.e. the change does not require further, potentially extensive changes throughout the whole analysis workflow.

Figure 5.15 illustrates how different analysis steps typically build up on each other. This information was derived by examining the components from the DKPro Core collection. While we note, that most of the time the order of analysis steps is the same across tools from different vendors, there are also some exceptions:

- Stanford CoreNLP requires tokenization before sentence splitting. Where this is not required, we place the tokenization step after the sentence splitting step, to avoid tokens spanning across sentence boundaries, e.g. because an abbreviation was not properly detected. DKPro Core users are not affected by this difference, because tokenizing and sentence splitting are always integrated into a single *segmenter* component.

- Stanford CoreNLP derives the dependency relations from the constituent structure using rules. Pure dependency parsers build directly on part-of-speech and token information.
- The Mate-tools use lemma information as input to the part-of-speech tagger (cf. [5]), while more often, the part-of-speech tag is used to take into account the context of a token and to disambiguate the lemma, e.g. by looking it up in a morphological database.
- The Mate-tools dependency parser can use lemma information for better results.

We also need to note here, that, so far, there is only one component for semantic role labeling and one for coreference resolution integrated into DKPro Core. Hence, the dependencies for these tasks are based only on these components. In general, we should assume that these steps basically build up on any of the previously generated information. E.g. named entity information could also be useful for semantic role labeling, while dependency relations may be useful for coreference resolution.

The order in which the analysis steps need to be performed is not always the same, depending on the analysis components being used. This affects interchangeability, because replacing one component for another may require changing the order of the components in an analysis workflow. However, based on the examined components, no significant flow incompatibilities could be found.

5.2.4.2 Conceptual interoperability

Conceptual interoperability is interoperability at the level of annotation types and tag sets. Annotation types were already discussed in Section 5.1. We now examine if components in the DKPro Core collection are interoperable on the level of tag sets for parts of speech, constituents (syntactic categories), and dependencies (syntactic functions).

Interoperability on the tag set level can be a major problem for a broad-coverage collection. Since the analysis components and their models come from many sources, there is a risk that each source uses their own tag sets and the components would not be interoperable, or possibly only on the limited level provided by some coarse-grained tags used as elevated types.

We previously noted (Section 5.1), that a mapping of the tags produced by individual analysis components to coarse-grained types may be used to make components interoperable to a certain degree, in particular across different languages. This mapping usually incurs a loss of information. It is preferable, in particular when operating within a single language, that all components use the same tag sets.

We added the ability to record tag set information as part of the analysis results to many of the components in DKPro Core. In most cases, the tag set information is extracted directly from the models[12] used by an analysis component or sometimes from the analysis component itself, e.g. when the tags are part of a hard-coded set of rules. Additionally, a tag set name is recorded, if such a name has been provided as part of the model metadata that DKPro Core adds in its proxy artifacts (cf. Section 3.1). In most cases, we tried to guess which tag sets are used by a particular model, taking into account any documentation we found for the respective models. Based on this, we started setting up mappings from these tag sets to coarse grained

[12] We have collected these models from many sources. For the most part, they are packaged with the respective analysis tools or are available from their websites. However, some of them have also been created by unrelated third parties and were made publicly available on the internet.

elevated types, so far mainly for part-of-speech tag sets. However, for future versions of DKPro Core, we consider setting up such mappings also for other layers, e.g. for dependency relations (cf. [151]).

Since there is no canonical inventory of tag sets, we typically used an abbreviated name of the corpus for which the tag set was developed (e.g. *ptb* for Penn Treebank, *ctb* for Chinese Treebank, etc.) but also other names if convenient. To our knowledge, the currently most complete resource on tag sets is the *Ontologies of Linguistic Annotation* (*OLiA*) [43]. Where applicable, we provided references to this ontology, but we will also see that it covers not even half of the tag sets we encountered.

Part-of-speech tags

Part-of-speech tags are produced or consumed by many analysis components. Table 5.7 provides an overview over the part-of-speech tag set mappings we use for models for analysis components integrated into the DKPro Core component collection for various languages.

Based on this overview, it appears that the interoperability between analysis components using part-of-speech information is pretty good for several languages, e.g. for English, German, Arabic, Chinese, and Bulgarian. In all these languages, there is a predominant part-of-speech tag set. For the other languages, in particular for Spanish and French, we observe a variety of different tag sets.

As Petrov et al. [178] note, interoperability between part-of-speech taggers producing different tag sets and higher level analysis components can be created by mapping the tag sets to a set of coarse grained tags. However, this requires the higher-level analysis to be trained on the coarse grained tags. So far, we found only a set of Spanish models for the OpenNLP part-of-speech tagger[13] producing these universal tags – no parsers or other components. Although it would be easy to create such models for researchers having access to corpora which can be converted to the universal part-of-speech tag set, the adoption in the community currently still appears to be low. This might change if the universal dependency tag set proposed by McDonald et al. [151] becomes popular. The example data they currently provide uses the universal part-of-speech tags as a coarse grained tag set.

Part-of-speech annotations created by the DKPro Core components contain the tag of the original tag set in the *posValue* feature, while there is a set of coarse grained *elevated* annotation types which largely mirror the universal tags (cf. Section 5.1). Due to user requirements, DKPro Core makes a distinction between common nouns (*NN*) and proper nouns (*NP*), which is not made by Petrov et al. However, since these use the simple noun category (*N*) as a common supertype, this is a fully compatible extension of the universal tags.

Constituency and dependency parsing

For constituency parsing (Table 5.8) and dependency parsing (Table 5.9), fewer tools were integrated into DKPro Core. For constituency parsing, there appear to be no interoperability problems, as there is only one tag set used for each language we found models for. For dependency parsing, the situation is more heterogeneous. For English and Spanish, the parsers use different tag sets. For German, Swedish, and Chinese, we only found one publicly available model each, so a comparison is not possible.

We should mention at this point, that research on dependency parsing is quite active. There are various publications on the adaptation of the Stanford dependency relations [57] to other languages than English, e.g. to Chinese [42], Italian [31], or Hebrew [221]. There are even considerations for establishing a set of universal dependency relation categories that can be used across languages [151].

[13] Part-of-speech tagger models for Spanish trained on the universal part-of-speech tag set: `https://github.com/utcompling/OpenNLP-Models` (Last accesses: 2013-10-06)

Table 5.7: Part-of-speech tag sets

Component	ar	bg	da	de	en	es	fr	it	nl	pt	sv	zh
Part-of-speech taggers												
MatePosTagger	–	–	–	stts	ptb	ancora	melt	–	–	–	–	ctb
ClearNlpPosTagger	–	–	–	–	ptb	–	–	–	–	–	–	–
OpenNlpPosTagger	–	–	ddt	stts	ptb	parole-reduced, universal	–	tanl	alpino	bosque	talbanken05	–
StanfordPosTagger	atb	–	–	stts	ptb	–	–	–	–	–	–	–
TreeTaggerPosLemmaTT4J	–	btb	–	stts	ptb	crater	stein	stein	tt	gamallo	–	lcmc
Parsers												
BerkeleyParser	atb	btb	–	stts	ptb	–	ftb	–	–	–	–	ctb
MaltParser	–	–	–	–	ptb	freeling	melt	–	–	–	stb	–
MateParser	–	–	–	stts	ptb	ancora	melt	–	–	–	–	ctb
OpenNlpParser	–	–	–	–	ptb	–	–	–	–	–	–	–
StanfordParser	atb	–	–	stts	ptb	–	ftb	–	–	–	–	ctb

Tag set ID	Reference	OLiA Reference [43]
alpino	Van Der Beek et al. [224]*	–
bosque	Freitas and Afonso [93](Anexo 4)	–
atb	Bies [23], [142]*	–
ancora	Taulé et al. [212]	–
btb	Osenova and Simov [175]	–
crater	Sánchez León [188]	–
ctb	Xia [228]	`http://purl.org/olia/pctb.owl`
ddt	Kromann et al. [134]	–
freeling	PAROLE-ES [176]	`http://purl.org/olia/parole_es_cat.owl`
ftb	Abeillé and Clément [1]	`http://purl.org/olia/french.owl`
gamallo	Gamallo [95]*	–
melt	Crabbé and Candito [49]	–
lcmc	McEnery and Xiao [152]	–
parole-reduced	PAROLE-Reduced [177]	–
ptb	Santorini [189]	`http://purl.org/olia/penn.owl`
stb	Ejerhed et al. [76]*, [202]	–
stein	Stein [203]	`http://purl.org/olia/french-tt.owl`
stts	Schiller et al. [191]	`http://purl.org/olia/stts.owl`
talbanken05	MAMBAlex [145]	–
tanl	TANL-IT [211]	–
tt	TT-NL [222]	–
universal	Petrov et al. [178]	–

Note: Citations marked with an asterisk (*) may only cover the research context, tool, or resource in which the tag set is used, but may not provide an explicit account of the tags used in the tag set.

In the current DKPro Core version 1.5.0, we provide elevated types for dependencies and constituents, but these are not coarse grained universal categories. Rather, for constituents, the Penn Treebank tag set and for dependencies, the Stanford tag set is used. As the state of the art advances, we consider adopting coarse grained elevated types for constituents and dependencies as well.

Interoperability issues

Although there is some heterogeneity of tag sets within individual languages, our analysis of publicly available models leads us to believe there is a good degree of interoperability. However, components may still not be interoperable – or at least a decrease in quality is to be expected – if certain components or models are replaced by others. Sometimes, two corpora are annotated with very similar tag sets, but different annotation guidelines (cf. [57] vs. [46]). When

Table 5.8: Constituent tag sets

Component	ar	bg	de	en	fr	zh
BerkeleyParser	atb	btb	negra	ptb	ftb	ctb
OpenNlpParser	–	–	–	ptb	–	–
StanfordParser	atb	–	negra	ptb	ftb	ctb

Tag set ID	Reference	OLiA Reference [43]
atb	Maamouri et al. [143]	–
btb	Osenova and Simov [175]	–
ctb	Xue et al. [230]	–
ftb	Abeillé et al. [2]	–
negra	Brants et al. [33]	`http://purl.org/olia/tiger.owl`
ptb	Bies et al. [24]	`http://purl.org/olia/penn-syntax.owl`

Table 5.9: Dependency relation tag sets

Component	de	en	es	fr	sv	zh
ClearNlpDependencyParser	–	clear	–	–	–	–
MaltParser	–	stanford	iula	ftb	stb	–
MateParser	negra	conll2008	ancora	ftb	–	cpb
StanfordParser	–	stanford	–	–	–	–

Tag set ID	Reference	OLiA Reference [43]
ancora	Soriano et al. [199]	–
conll2008	Surdeanu et al. [208]	–
clear	Choi and Palmer [46]	–
cpb	Xue [229]	–
ftb	Candito et al. [39]	–
iula	Marimon et al. [147]*	–
negra	Brants et al. [33]	–
stanford	de Marneffe and Manning [57]	`http://purl.org/olia/stanford.owl`
stb	STB-DEP [201]	–

Note: Citations marked with an asterisk (*) may only cover the research context, tool, or resource in which the tag set is used, but may not provide an explicit account of the tags used in the tag set.

models are created based on such corpora and used in analysis components, the results appear equivalent to the user. However, consider a rule-based information extraction system based on dependencies. Such a system is likely to perform badly when applied to a dependency structure which uses the same tags as the rules, but was created based on different annotation guidelines that assumed by the rules.

Most of the DKPro Core components have the capability of extracting tag set information directly from the models they have been configured with. We used this capability to compare the dependency tags of models for several dependency parsing components (Table 5.10) to find clues which may indicate that the models are based on different annotation guidelines. The table shows only those tags where there are differences between the examined models.

We take the tag set used by the Stanford parser as a point of reference. The Stanford parser uses a rule-based conversion of the constituent tree into dependencies, so the tag set is in fact not part of the models – it is extracted directly from the tag set known to the parser. The other two parsers, MaltParser and the ClearNLP parser, are statistical parsers which generate dependency structures based on text annotated with part-of-speech tags. If a Stanford tag is missing from their tag sets, it can be either because the corpus they were trained on did not

Table 5.10: Differences between dependency tag sets

Component	Variant	ROOT	abbrev	agent	arg	comp	complm	cop
ClearNlpDependencyParser	mayo			agent			complm	
ClearNlpDependencyParser	ontonotes			agent			complm	
MaltParser	linear	ROOT	abbrev				complm	cop
MaltParser	poly	ROOT	abbrev				complm	cop
StanfordParser	factored			agent	arg	comp		cop
StanfordParser	pcfg			agent	arg	comp		cop

Component	Variant	csubjpass	discourse	goeswith	gov	hmod	hyph	intj
ClearNlpDependencyParser	mayo					hmod	hyph	intj
ClearNlpDependencyParser	ontonotes	csubjpass				hmod	hyph	intj
MaltParser	linear	csubjpass						
MaltParser	poly	csubjpass						
StanfordParser	factored	csubjpass	discourse	goeswith	gov			
StanfordParser	pcfg	csubjpass	discourse	goeswith	gov			

Component	Variant	measure	meta	mod	mwe	nmod	npadvmod	null
ClearNlpDependencyParser	mayo		meta			nmod	npadvmod	
ClearNlpDependencyParser	ontonotes		meta			nmod	npadvmod	
MaltParser	linear	measure						null
MaltParser	poly	measure						null
StanfordParser	factored			mod	mwe		npadvmod	
StanfordParser	pcfg			mod	mwe		npadvmod	

Component	Variant	obj	oprd	pred	purpcl	ref	rel	root
ClearNlpDependencyParser	mayo		oprd					root
ClearNlpDependencyParser	ontonotes		oprd					root
MaltParser	linear			pred	purpcl		rel	
MaltParser	poly			pred	purpcl		rel	
StanfordParser	factored	obj		pred		ref	rel	
StanfordParser	pcfg	obj		pred		ref	rel	

Component	Variant	sdep	subj	tmod	xsubj
ClearNlpDependencyParser	mayo				
ClearNlpDependencyParser	ontonotes				
MaltParser	linear			tmod	
MaltParser	poly			tmod	
StanfordParser	factored	sdep	subj	tmod	xsubj
StanfordParser	pcfg	sdep	subj	tmod	xsubj

Note: This table shows only those tags where there are differences between the examined models. Tags that are shared by all models have been omitted. The MateParser does not appear in this overview, because its English model uses the CONLL dependency tag set, which is not related to the Stanford tag set.

contain a corresponding dependency relation (coverage), or because the corpus used different annotation guidelines which omitted the tag. However, if there is a tag which is not present in the Stanford tag set, it is a good indicator that different annotation guidelines have been used, which may cause further incompatibilities.

For the MaltParser models, the tags *abbrev* and *purpcl* are documented in the Stanford dependency manual [57]. However, they are not present in the version of the Stanford parser we used.[14] The tag *measure* is not documented in the manual. The tags *null* and *ROOT* are likely to be artifacts of the parser implementation and not actually present in the training data.

For the ClearNLP parser models, we note several new tags which are not covered by the Stanford manual: *hmod*, *hyph*, *intj*, *meta*, *nmod*, *oprd*. There are also multiple tags present in the MaltParser models and in the Stanford manual, which are not used by the ClearNLP models. In fact, the ClearNLP models have been trained using different annotation guidelines,

[14] The parser has been taken from Stanford CoreNLP [200] version 3.2.0, and the inspected class is `EnglishGrammaticalRelations`.

which build up on many of the Stanford tags, but add also new ones. Looking only at the tag sets, at first it appeared just as if the ClearNLP models were using a richer tag set and that the training data did not cover certain kinds of constructions. Fortunately, following an e-mail conversation with the ClearNLP developer Jinho Choi, he put the annotation guidelines [46] online which had previously not been available. From the guidelines, we could then learn that the structure of the dependency annotations is different. Based on this information, we then changed the tag set metadata for the ClearNLP parser models from what we previously believed to be the Stanford dependency tag set (*stanford*) [57] to the CLEAR dependency tag set (*clear*).

Noh and Padó [166] report on building on DKPro Core within the EXCITEMENT Open Platform for textual entailment.[15] While they note to benefit substantially from the provided interoperability, they also note that there are implicit dependencies between analysis tools. In their system for textual entailment, they use rules over the output of a dependency parser. As these rules were created based on dependency parsing that follow certain annotation guidelines, exchanging the dependency parser in their workflow for one that produces dependency relations based on different guidelines causes problems, because the rules no longer match.

Desiderata

While previous research on interoperable analysis components has noted these incompatibilities at the conceptual level, interoperability appears to be mostly discussed as being a problem of the naming of types and features (e.g. [179; 118; 113]). Because in DKPro Core, all components used the same type system, there is no need for extensive mapping and normalization between different representations. We concentrate on analysis steps that are covered by analysis tools from multiple vendors and note that these tools consume and produce information that does not require very elaborate annotation types (segmentation, part-of-speech tags, lemma, constituency parsing, dependency parsing, etc.). The only level of normalization that we currently perform over the raw output of the tools are the coarse grained elevated types.

We notice that just by looking at the tags provided by a model, it is sometimes difficult to tell to which tag set and underlying annotation guidelines these tags belong. For example, the CLEAR dependency tag set [46] and the Stanford dependency tag set [57] largely overlap in terms of tags, but prescribe different structures in their annotation guidelines. The annotation guidelines used to prepare the training data or to create the transformation rules contain important information about the assignment of tags and the interaction between tags. However, when models are distributed, exact information about which annotation guidelines were used in the form of a versioned reliable reference is often not provided. Neither is an exact versioned reference to the training data provided in most cases. Given the present standards, a model can already be considered well-documented if it provides a non-versioned reference to the training data, e.g. in form of citation or link to a website.

Based on the above analysis, we derive the following recommendations:

- A publicly available portable workflow (cf. Section 4.1) should exist for each model, which could be used by anybody with access to the necessary resources to regenerate the model.
- Annotated corpora (along with their annotation guidelines) should be made available via a repository, so that they can be referenced and accessed by such workflows.
- There should be a way to fetch a versioned reference to the guidelines used to annotate the data from which a model was generated directly from the model file, e.g. via a reference to ISOcat [130], OLiA [43], or an equivalent resource.
- At least the annotation guidelines for corpora should be made publicly available, even though this may not be possible for the annotated corpora themselves due to legal restrictions.

[15] http://www.excitement-project.eu (Last accessed: 2013-12-11)

5.2.4.3 Provenance and attribution

Processing frameworks and convenient component collections place a layer of abstraction between the users and the components and resources being used. We notice that users start talking about *using DKPro* or *using UIMA* and tend to not mention anymore which particular underlying analysis tools and models have been used, e.g. *the Stanford PCFG parser*. This may be due to the rich choice and the convenience of automatic resource selection which make it increasingly difficult to determine what to cite when publishing new results since the link to the source work tends to be lost. This is damaging in many ways. Users may not critically reflect if and why a certain tool or resource is suitable for their task. Providers of resources and tools may be set back by their work being used without proper attribution.

Provenance

Knowing about provenance, i.e. knowing which work is involved in an experiment and contributes to the generation of the experimental results, is a prerequisite for attribution. This not only extends to the components and resources were involved in the creation of analysis results or which configuration led to the creation of this result (cf. [84]). Also, the knowledge about how particular resources being used in a workflow were created in the first place, e.g. on which corpora statistical models were trained and which annotation guidelines were used for the annotation of these corpora, is important. In order to be able to comply with the requirements towards attribution, the researcher must know about the provenance of the results that an experiment has produced.

Attribution

Another problem is attribution of third-party work. Attribution of other people's work and giving them credit when their work is incorporated into follow-up work is an important issue. In software development, this is covered by certain files (e.g. *NOTICE*, *COPYRIGHT* or *LICENSE*), which are shipped with a library and which must be preserved and reproduced when the library is used in a larger work. For the components and resources used in an analysis workflow, their authors also often require to be mentioned with references to their respective work, typically in the form of publications which describe a particular algorithm or resource.

Desiderata

Provenance and attribution are not new issues, particularly being discussed in domains that deal primarily with content creation (e.g. arts) and in the context of workflow engines in general (e.g. [92]). However, in the context of processing frameworks and analysis component collections, these questions are not well addressed yet. We believe this is an important shortcoming that should receive more attention by processing frameworks and component collections used for language analysis.

In future versions of the DKPro Core component collection, we plan to maintain attribution information for each integrated tool and resource and to expose this information along with other analysis results, such that the third-party work involved in the creation of these results can easily be identified and cited by the user. This way, we aim to restore the awareness that it is important to report which of the integrated tools and which resources were used, despite the convenience of a component collection.

It also needs to be investigated if the adoption of a general provenance model, e.g. the *Open Provenance Model* (*OPM*) [159] is beneficial. OPM is an abstract model for representing the provenance of a thing as a directed graph. Nodes in such a graph represent *artifacts* (immutable states), *processes* (actions producing new artifacts), and *agents* (entities involved in a process). Edges represent causal dependencies, e.g. *used*, *wasGeneratedBy* or *wasControlledBy*.

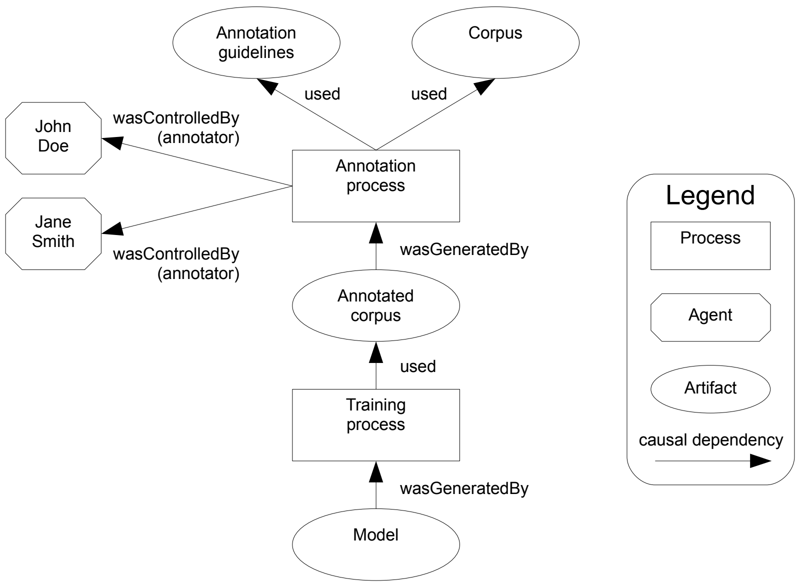


Figure 5.16: Provenance of a model represented as an OPM graph

Contrary to a workflow, which *can be used* to generate an artifact in the future, the OPM graph is a model of the past history of how an artifact *has been produced*. A benefit of adopting such a provenance representation would be the ability of aggregating provenance fragments for models, components and primary data during the execution of a workflow to generate a full provenance model of the workflow results. An illustration of how OPM could be used to represent the provenance of a model is given in Figure 5.16.

5.2.5 Summary

In this section, we have examined the DKPro Core broad-coverage collection of interoperable analysis components. At the time of writing, we found DKPro Core to be the most comprehensive collection of portable analysis components, i.e. components that do not rely on web services for processing. The analysis tools integrated in DKPro core come from different research contexts focusing on different languages and domains. Thus, DKPro Core provides the users with a rich choice and increased coverage.

In the current edition, we also focused on improving the usability of DKPro Core. We have incorporated the automatic resource selection and acquisition mechanism introduced in Section 3.1. We also have introduced a set of common parameters across the component collection to ensure that users do not have to learn a completely new set of parameters every time they use a new component.

Whether a user can conveniently exercise the choice offered by DKPro Core depends on the interchangability of the analysis components. We have analyzed the components and found most of them that serve for the same analysis task, e.g. part-of-speech tagging or parsing, to be consuming the same input and producing the same output. They are therefore interchangeable within the workflow.

At the conceptual level, interoperability depends on the tag sets and on the annotation guidelines. Our analysis of the tools and their models integrated in DKPro Core has shown that there exists a good interoperability between the components for languages which receive a lot of research attention, such as English and German. The community has converged on certain tag sets and provides the necessary models for these languages. For other languages, there we have found various tag sets to be in use, thus, they are not directly interoperable. We have found that tag sets may appear to be largely similar, but even small differences are clues that they in fact use different underlying annotation guidelines.

We are happy to report that DKPro Core is no longer only the foundation of research at the Ubiquitous Knowledge Processing Lab. It is now also being recognized and used externally, e.g. by the EXCITEMENT Open Platform for textual entailment [78; 167], by Riedl and Biemann [185] in an experiment on text segmentation with topic models, in the JoBimText project[16] [125; 102], and by Strötgen and Gertz [207] in experiments on temporal tagging. We believe this to be an effect of our continuous efforts to increase the coverage of DKPro Core and to improve its usability, such as they have been documented in this thesis.

In future work, we plan to include more metadata in analysis results. In particular, we intend to include provenance information with resources and analysis components. This will allow users to easily identify and give attribution to third-party work used to produce their analysis results.

[16] According to a code search engine:
`http://code.ohloh.net/search?s=%22de.tudarmstadt.ukp.dkpro.core%22` (Last accessed: 2013-12-11)

6 Interactivity

In the previous sections, we have addressed the integration of manual and automatic analysis mainly by making automatic analysis easier to use and more accessible to users who so far work predominantly manually and who are not programming experts. In this section, a more direct form of integration is addressed, where users can interactively work with automatically generated analysis results and manually correct these or add new results that could not be automatically created.

Roles

We first take another look at the four roles involved in the manual analysis tasks (cf. Section 2.1) and elaborate on them:

- **Explorer** – This role performs preliminary exploration of the corpus data, either to generate linguistic hypotheses or to corroborate already existing hypotheses. A preliminary exploration is usually done before the first version of the annotation guidelines is created.

- **Guideline author** – This role defines the annotation guidelines to be used, including the tag sets. It requires expert domain skills, e.g. in the linguistic domain. A guideline author, in most cases, first acts as an explorer.

- **Annotator** – This role performs manual analysis based on the annotation guidelines. Basic domain skills are required to interpret the annotation guidelines. For example, undergraduate students can often take on this role. If the annotation guidelines are extremely simple, no special skills may be required and the annotation task can be crowdsourced. In some cases, the annotation guidelines are very complex, so that the annotator may require a skill level similar to that of the explorer role.

- **Curator** – This role critically examines the annotations to resolve cases where the annotators did not agree. This can lead to recommendations to the guideline author to improve the annotation guidelines and to another iteration for the annotators. When a domain expert takes on this role, e.g. the same person as the guideline author, the curator may also directly generate a curated annotation which is then supposed to be *correct*. The role can also be assumed by an automatic mechanism that accepts a label assigned by a sufficient number of annotators as the correct one, e.g. via a majority vote.

These roles mostly follow the ones defined by Dipper et al. [60]. The *language engineer*, applying automatic methods to the corpus, is left out here, as it is better described by the roles for automatic analysis (see Section 2.2). The *guideline explorer*, interested in the principles underlying the annotation guidelines, is not relevant to the further discussion here.

The *annotator* and *curator* roles are also related to those defined by Bontcheva et al. [29]. Furthermore, they conflate the *guideline author* role into the *curator* role. No equivalent of the *explorer* role is mentioned. Additionally, a *project manager* role is defined, which has mainly technical and administrative responsibilities, such as setting up an annotation project in an analysis tool, providing the corpus, overseeing the project progress, etc.

Annotation activities

In a basic manual analysis setup, four main activities can be distinguished (Figure 6.1): *exploration*, *writing of annotation guidelines*, *annotation*, and *curation*. The fifth activity shown in the diagram, *bootstrapping*, is usually an automatic analysis task.

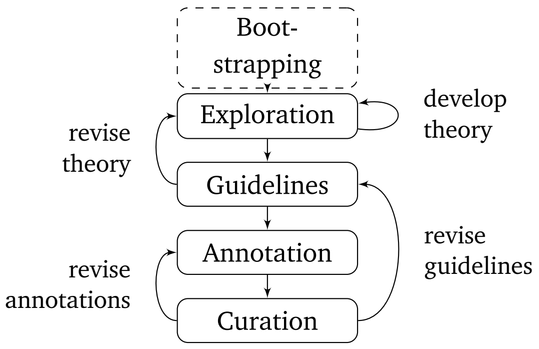


Figure 6.1: Basic manual analysis

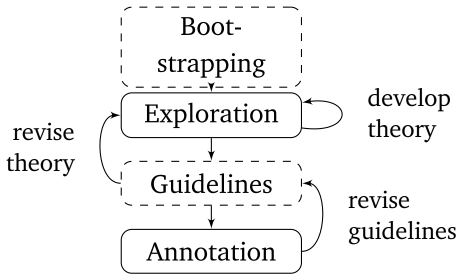


Figure 6.2: Preliminary manual analysis

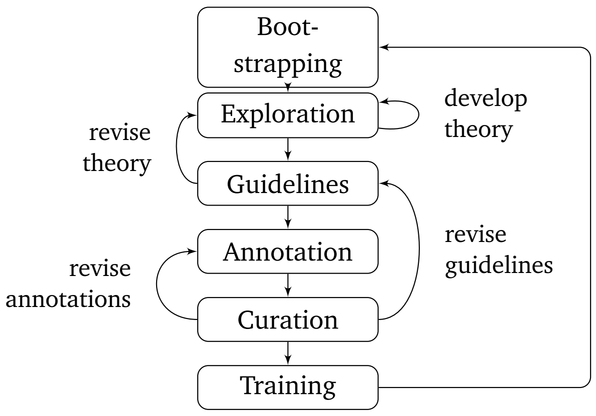


Figure 6.3: Manual analysis with automatic bootstrapping and re-training

The roles are closely related to these activities. Some of the activities are done in phases, not parallel to each other, so that one role remains idle until another role has completed its work, in particular the *writing guidelines*, *annotation*, and *curation* phases. As will be pointed out below, the *exploration* and *writing guidelines* phases often closely interact with each other.

- **Bootstrapping** – During the bootstrapping phase, the data is prepared by applying some initial linguistic analysis to the plain text corpus. This initial analysis is usually done automatically, which is why this is not considered one of the main activities of manual annotation here. The initial annotations can basically serve two purposes:
  - **Correction** – the annotations are examined and corrected with the goal of re-training and improving automatic analysis applied during the bootstrapping phase in a subsequent iteration (cf. Figure 6.3);
  - **Cues** – the annotations are used as cues for higher-level annotations, e.g. part-of-speech tags are a prerequisite for the annotation of dependency relations. The initial annotations may not even be editable in such a scenario. Annotation projects aiming at very high quality may choose to build on a previous manual annotation task for bootstrapping, instead of relying on automatic analysis.
- **Exploration** – The exploration phase is mainly used to generate hypotheses and to gather evidence to corroborate or invalidate the hypotheses. It may also be used to gather initial examples that can be used for writing the annotation guidelines and to generate an initial set of categories to be used later for annotation.

- **Writing of annotation guidelines** – Annotation guidelines explain how to identify text spans that should be labeled and how to decide which tags should be used for labeling. They need to be prepared before annotators can be trained and before they can start annotating. As Dipper et al. [60] point out, the guidelines need to serve different roles, such as the *annotator*, the *explorer*, etc. For example, while the *annotator* should be able to perform the annotation task efficiently based on superficial annotation guidelines, the *explorer* and later the *curator* may require in-depth descriptions of the theoretical underpinning of each category and the decisions taken in their construction. As Dipper et al. [59] point out, the writing of guidelines and annotation activities affect each other. In particular, while writing the initial version of the guidelines, inconsistencies in the theory or the tag set become obvious, so it becomes necessary to switch often between the *exploration* and *writing of annotation guidelines* activities. Later, changes to the annotation should be fewer and be well separated from the annotation phase to avoid wasting resources by having the annotators repeatedly revise all data already analyzed in the current iteration and update it to changes in the guidelines. As Benikova et al. [18] point out, it can be helpful to exchange the annotation team when the annotation guidelines have stabilized, in order to avoid influences of earlier versions of the guidelines on the annotators which may lead to erroneous annotations and low inter-annotator agreement.

- **Annotation** – The actual analysis and main effort take place during the annotation phase. Typically, this phase can only commence once annotation guidelines are available. However, during a preliminary analysis (cf. Figure 6.2), e.g. conducted by a single researcher to set the groundwork for an annotation project in a larger group, this task can go hand-in-hand with the *exploration* task and towards the end of the analysis leads to *writing the annotation guidelines*. In projects aiming to attain high-quality annotations, the data is usually distributed among the annotators in such a way, that every unit of annotation (e.g. every document or sentence) is analyzed by a minimum number of *annotators*. To avoid bias, it may also be a requirement, that each annotator produces the analysis solely based on the annotation guidelines, without discussing the process or otherwise interacting with fellow *annotators*. The annotation process can be supported by a mechanism suggesting the most probable locations for an annotation or the most probable tags. However, unless such assistance is almost indispensable, for example because the corpus to annotate is very large with relevant passages being sparsely distributed within the corpus, projects may prefer rejecting it, because it biases the *annotators* decisions.

- **Curation** – The curation phase aims at aggregation and quality control. Aggregation is necessary to produce a single final analysis result from the separate analysis result that each *annotator* has produced during the *annotation* phase. The aggregation can be done manually, by a human *curator* reviewing all the annotations and selecting those considered *correct*. The *curator* may be supported by the software, which performs a preliminary comparison of the individual analysis results and highlights those annotations that the annotators did not agree upon. The aggregation can also be done fully automatically by applying a voting mechanism to decide what remains in the final result. Quality is ensured by the standards that are applied during the aggregation. E.g. a minimal level of agreement between the *annotators* may be required to accept an annotation into the final result (cf. [16]).

Extended manual analysis workflows may consist of many more states, but should generally follow the scheme outlined above. For example, for the task of building a reference translation corpus, Friedman et al. [94] ensure quality using a cascade of *six(!)* translation revision steps, performed by translation experts with different levels of qualification instead of the *annotation* and *curation* steps.

A manual analysis project usually requires several iterations of the full analysis process, with early iterations being shorter and following a simplified process, such as the preliminary steps outlined in Figure 6.2, leading to a more refined process in later iterations, once the annotation guidelines have been stabilized.

If the ultimate analysis is rather complex, e.g. the analysis of the full dependency structure, multiple iterations of the process may be used to annotate only certain tags at a time, e.g. first all subjects, then all direct objects, etc. This requires the *annotators* to keep fewer guidelines in mind during each iteration and to perform the analysis faster and more reliable (cf. [16]). Annotation editors can support this as a mode of operation and offer a simplified annotation interface allowing for faster annotation, such as the *fast annotation mode* of Knowtator [170] or the crowdsourcing mode supported by Yimam et al. [232].

6.1 Search and annotation

In this section, we present the *annotation-by-query* approach to annotating infrequent phenomena in large corpora. First, we discuss how to employ search technology to locate and annotate infrequent phenomena in large, pre-annotated corpora. We examine existing tools and find that they either support search on large corpora or annotation, but not both. To address this issue, we define an annotation process which covers both, search and annotation. The process requires new kinds of interactions between the members of the annotation team, which a tool supporting the process needs to provide. Next, we introduce a new tool supporting the process. Finally, we discuss how the resulting tool has been used in linguistic research.

These contributions address the following issues in our overall scenario (Figure 6.4):

❻ **There are no adequate tools for the selective annotation of large corpora.**
The tight integration of linguistic search engines with an analysis tool, as offered by our *annotation-by-query* process, enables the selective annotation of large corpora by integrating the *explorer* and *annotator* roles. Additionally, the process is defined in such a way, that an annotator team can work fully distributed on the selective annotation task. To support our process, we introduce the CSniper annotation tool. CSniper additionally supports the *guideline author* role by allowing the user to maintain the annotation guidelines directly in the tool. The *curator* role is taken by an automatic aggregation mechanism, which allows the annotation team to generate an aggregated result on demand, based on configurable quality settings.

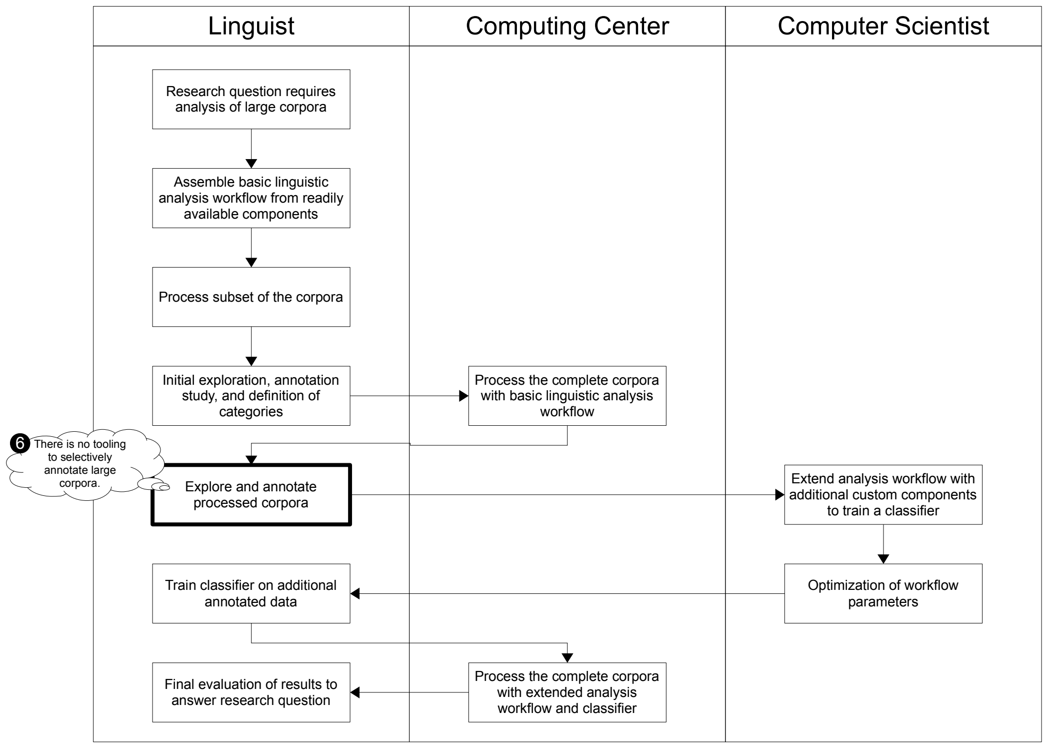

Linguist
Computing Center
Computer Scientist
Research question requires analysis of large corpora
Assemble basic linguistic analysis workflow from readily available components
Process subset of the corpora
Initial exploration, annotation study, and definition of categories
Process the complete corpora with basic linguistic analysis workflow
❻ There is no tooling to selectively annotate large corpora.
Explore and annotate processed corpora
Extend analysis workflow with additional custom components to train a classifier
Optimization of workflow parameters
Train classifier on additional annotated data
Process the complete corpora with extended analysis workflow and classifier
Final evaluation of results to answer research question

Figure 6.4: The tool CSniper supports the selective annotation of large corpora by implementing the *annotation-by-query* process.

6.1.1 Motivation

Many analysis tools assume that a corpus will be exhaustively annotated and that the annotators will read the full corpus, e.g. document by document or sentence by sentence. However, when only specific phenomena are to be annotated and when these occur infrequently in corpora, new approaches to annotation are required. In fact, finding these phenomena in the first place can already be a challenging task:

> *However, for the annotators, the corpus is often a haystack within which they must find what to annotate, and discriminating what should be annotated from what should not is a complex task.* *Fort et al. [88]*

Annotations already present in the corpus under examination can help the researcher to locate passages containing the desired phenomena. This calls for annotation editors which not only support annotation, but also searching in a pre-annotated corpus in order to locate and selectively annotate only relevant passages.

Selective annotation using a combination of a linguistic search engine and an annotation editor has been addressed in the past. The *SALTO* tool [36] integrates the *TIGERSearch* [138] engine for queries over treebanks. It is reported that this search functionality was used during the SALSA project [37] to annotate role-semantic information lemma-by-lemma.

We reported on a similar procedure in Eckart de Castilho et al. [74]. It was used by Teich and Holtz [214] for a study on how certain concepts behave in different scientific domains based on a corpus of research papers from nine different domains. For example, the word *algorithm* takes on different roles in computer science and in biology. In order to conduct this research, passages containing certain words needed to be located and selectively annotated for *transitivity*, a linguistic concept from the Systemic Functional Grammar [111] used to describe processes, participants in these processes, and circumstances under which the processes occur or are performed. We did this by performing a search in the IMS Corpus Workbench [77], exporting the search results including some preceding and following context to a file, and transforming the file into the format of the UAM Corpus Tool [168] using the AnnoLab framework [69]. After the annotations had been made within the UAM Corpus Tool, the data was again transformed using AnnoLab and merged into an XML database containing the full corpus data.

However, in both tasks mentioned above, the selection took place before and separately from the annotation process. Also, the data selection process has been used to locate all occurrences of particular words or lemmata, which then have been annotated exhaustively.

In the course of this thesis, we worked on the task of finding non-canonical grammatical constructions in large corpora. A non-canonical construction (NCC) deviates from the canonical construction predominantly used in a language, e.g. the *subject-verb-object* order in English. For example, the ordering of information given by a sentence is changed from the default to focus on a particular fact (Figure 6.5).

1. *[S: The media] was now [V: calling] [O1: Reagan] [O2: the frontrunner].* *(canonical)*
2. *It was [O1: Reagan] whom [S: the media] was now [V: calling] [O2: the frontrunner].* *(non-canonical: it-cleft)*

Figure 6.5: Examples of a canonical and non-canonical variant of a sentence (cf. [75])

As NCCs appear with a relatively low frequency (cf. [17]), a corpus-based linguistic study requires analyzing a large corpus. Furthermore, some NCCs are ambiguous on the surface.

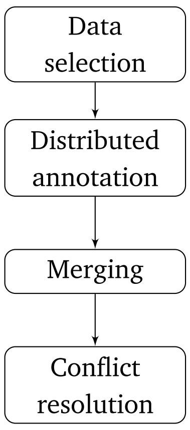


Figure 6.6: Partially distributed process used in *SALSA* [37]

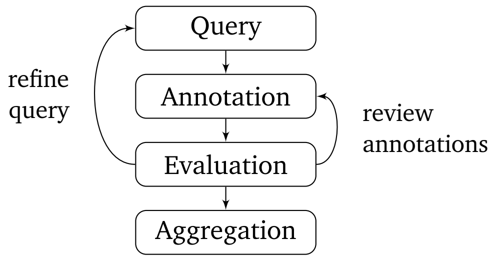


Figure 6.7: *Annotation-by-query* process (fully distributed)

Even for an expert, it is sometimes difficult to decide if a construction is non-canonical. A more detailed description of NCCs is given in Section 6.1.4. Our task was to develop a methodology and a tool by which the linguists could find as many non-canonical constructions with as little effort as possible.

Thus, we needed an approach in which searching is an integral part of the annotation process. Because we search for specific constructions which are ambiguous on the surface, not all search results would be proper matches. On the other hand, we did not require any sophisticated annotations. It was sufficient to annotate whether the sentences returned as the result of a search contained the construction in question or not. Due to our annotation team working in different locations, we required a fully distributed process (Figure 6.7). This means that every member of the annotation team can perform any of the steps independently from the other team members. This is different from the partially distributed process used while creating the SALSA corpus (Figure 6.6), where only the annotation step was distributed to the annotation team. First, all sentences with a specific predicate were selected. Then, all these sentences were annotated. In this way, the corpus was annotated one predicate at a time. The sentences were distributed to the annotators who annotated them independently. Their annotations were merged by an adjudicator who also resolved any conflicts.

In this section, we revisit the concept of integrating a linguistic search engine with an annotation editor [74] and define a new, fully distributed process for the annotation of ambiguous phenomena in large corpora. We also present a new tool called *CSniper* for distributed annotation supporting this process. Finally, the process and the tool are discussed in the context of our task of finding non-canonical constructions.

6.1.2 State of the art

In this section, we briefly examine existing tools and approaches which are useful for the identification of infrequent phenomena in large corpora. These cover in particular pre-annotating the corpus with an initial set of annotations that can be used for querying, tools for querying corpora, and different approaches to multi-user annotation.

6.1.2.1 Bootstrapping

Manually and exhaustively annotating a corpus is a tedious process. In the case that no adequately annotated corpus is available, automatically pre-annotating (bootstrapping) a corpus with automatically created annotations can facilitate the process by reducing the work of the human annotators to reviewing the annotations and correcting mistakes. A second use of bootstrapping is the creation of annotations which are not corrected, but used as a basis for higher-level annotations, e.g. to perform a linguistic search over the corpus, locating passages particularly interesting for the research task at hand. This second use is of particular interest in the present context.

GATE Teamware [30], for example, provides explicit support for bootstrapping. When an annotation project is set up, the tool allows setting up a workflow which mixes automatic analysis tasks with manual annotation tasks. Such a workflow may start with an automatic annotation task to bootstrap a corpus e.g. using a simple, rule-based approach. This could be followed by a manual annotation task, in which the results from the rule-based approach are corrected. The manually corrected data could be used by a third task to train a classifier which is then used to pre-annotate additional documents. In a final step, these additional documents could again be corrected manually.

The *UAM Corpus Tool* [168] also offers a simple form of bootstrapping with its *autocoding* feature. It allows the user to specify rules consisting of a pattern and a label. Whenever the pattern matches, the match is automatically annotated with the label.

6.1.2.2 Linguistic search

The ability to perform linguistic searches is provided by two kinds of tools: either as a convenience by annotation tools, or by dedicated linguistic search engines.

Search functionality in annotation tools

Some annotation tasks exhaustively analyze a given data set, e.g. scanning through every sentence in a corpus to mark named entities. This significantly limits the amount of data which can be analyzed by a human annotator. Assuming only one in a thousand sentences includes a named entity, this would be a very time-consuming approach. Thus, for infrequent phenomena, the ability to efficiently locate those passages which are relevant for annotation is an important aspect.

One of the analysis tools permitting the user to search through the corpus is *SALTO* [36]. The tool integrates the *TIGERSearch* [138] engine for searching over treebanks. It is reported that this search functionality was used during the SALSA project [37] to annotate role-semantic information lemma-by-lemma. In SALTO, the ability to perform a search and to distribute the results to the annotation team is limited to the admin role.

GATE Teamware also supports annotation in a distributed multi-user scenario. It integrates ANNIC [15] as a tool for searching corpora via JAPE patterns. However, this functionality is only available to the *manager* role in an annotation project, not to the annotators (cf. [79]).

Compared to this, we deal with a corpus which is too large for comprehensive annotation and in which the interesting phenomena are difficult to find. Additionally, we require that every member of the annotation team is able to perform searches.

Linguistic search engines

A linguistic search engine can be employed to search for the relevant passages. It works quite differently from a regular search engine used in information retrieval (IR) which locates documents and ranks them by relevance. For example, if a search term occurs only once in a large document, an IR search engine may rank it less relevant than a short document often containing the search term. Such a notion of relevance is usually not desired in an annotation task where a query is in fact a pattern, e.g. a regular expression. Thus, relevance is a binary decision as the annotated text either matches that pattern or not.

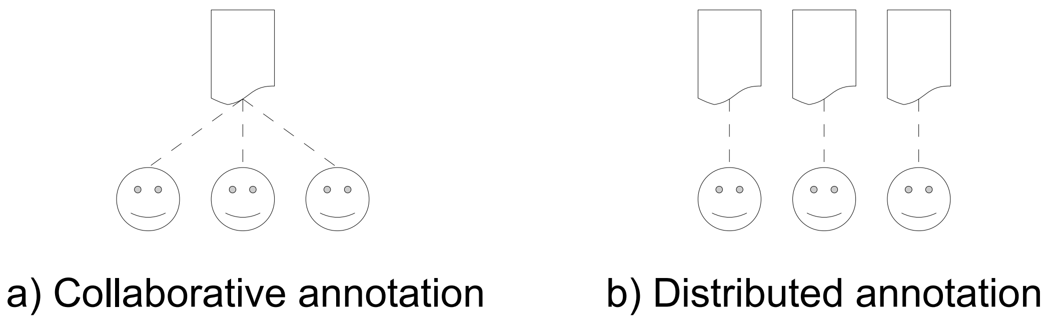


Figure 6.8: Modes of multi-user annotation

Various linguistic search engines exist, focusing on different kinds of annotated corpora. The *IMS Open Corpus Workbench* [77] is particularly well-suited for searching over corpora with annotations on the level of tokens, e.g. part-of-speech or lemma annotations. There is limited support for non-overlapping structural annotations above the level of tokens, like sentence or document boundaries. However, it is not at all suited for structures such as parse trees or dependency relations, neither in terms of the underlying technology nor in terms of the query language. *TGrep2* [187], *Fangorn* [101], or *TIGERSearch* [138] on the other hand target particularly the search over parse trees. These engines use specialized indexes to deal efficiently with dominance and sibling relations in parse trees and offer special operators for building concise queries over trees. Finally, ANNIS [233] supports a wide array of linguistically relevant structures and relations. It allows including multiple linguistic layers in a query, e.g. building a query with restrictions on the constituency structure, dependency relations, and tokens at the same time. To support this wide array of annotations, ANNIS had to make concessions on performance. TGrep2 or the IMS Corpus Workbench can operate comfortably on hundreds of millions of tokens, whereas ANNIS is limited to hundreds of thousands.

While these engines are well-suited for locating relevant passages in corpora, they do not allow immediately working further on these results, in particular to annotate these passages.

6.1.2.3 Multi-user annotation

In a multi-user annotation scenario, a group of annotators works together on a corpus. We distinguish between two modes of operation (Figure 6.8):

- **Collaborative annotation** – all annotators work on the same set of annotations.
- **Distributed annotation** – there is one set of annotations per annotator.

In both modes, the annotators work on the same set of documents.

Collaborative annotation

In a collaborative annotation scenario, all annotators work jointly on the same annotations. Interaction between the annotators is explicitly promoted and the work can be effectively shared between the annotators. However, the quality of the analysis cannot be assessed objectively, because there is no hard evidence if the annotators actually agree on the collaboratively created analysis results and double checked the results of their fellows.

Collaborative annotation does not appear to be widely used. The currently possibly most prominent collaborative analysis tool is *Brat* [204]. Most literature does not make an explicit distinction between collaborative and distributed modes of annotation. So much work referring to collaborative annotation actually describes a distributed annotation mode. However, at least since the advent of *Brat*, this distinction is an important one to make.

Distributed annotation

In a distributed annotation scenario, each annotator produces their own annotations. These can typically not be seen by fellow annotators to avoid unsolicited bias. Every annotator generally needs to put the same effort into the annotation task, as every one performs a full analysis of the corpus. The quality of the analysis can be assessed by comparing the results from different annotators with each other and calculating inter-annotator agreement.

Distributed annotation is common. Older analysis tools, e.g. SALTO [37], use client software and rely on a shared file system or database. Modern distributed analysis tools rely on web services, e.g. *GATE Teamware* [30], or are fully browser-based, e.g. *WebAnno* [232].

6.1.3 Contribution: An approach to the annotation of infrequent phenomena in large corpora

Existing approaches to distributed and collaborative annotation and the supporting tools are not well suited for the annotation of infrequent phenomena in large corpora. Some analysis tools provide support for automatically pre-annotating a corpus. However, these analysis tools do not offer sufficiently powerful search facilities to search large corpora and to interactively locate relevant passages. Linguistic search engines are capable of searching through large corpora, but they do not allow the users to annotate the search results.

To address the lack of a solution for annotating infrequent, ambiguous phenomena in a large corpus, we built up on the concepts presented by Eckart de Castilho et al. [74], and developed these ideas into the *annotation-by-query* process and the supporting tool *CSniper* [50; 75]. The approach and the tool distinguish themselves from others by addressing at the same time the *corpus explorer* role and the *annotator* role while operating on large corpora.

6.1.3.1 Process

The *annotation-by-query* process targets specifically the annotation of infrequent, ambiguous phenomena in large corpora. In contrast to a regular annotation process, not the annotation of documents but the search for relevant passages is at its core. The annotation is then performed on the search results only. It is this integrated approach to search and annotation which enables working with large corpora.

The process consists of four steps (Figure 6.7 on page 157): *query*, *annotation*, *monitoring*, and *aggregation*. While annotation, monitoring, and aggregation are standard steps for a distributed annotation process, the preceding *query* step as well as the *query refinement* are particular to the *annotation-by-query* process.

We opted for a distributed approach in which every annotator produces their own results, not for a collaborative approach where all annotators work on a common set of results. Consequently, an aggregation step to consolidate the annotations in a final analysis result is included in the process, as well as the possibility to calculate inter-annotator agreement. The query-based approach requires new solutions for reviewing annotations and to ensure that there is a sufficient number of overlapping annotations to calculate inter-annotator agreement.

Because every member of the annotation team can perform all steps independently from the other team members, we call our processes a *fully distributed* approach. This is in contrast to the *partially distributed* SALSA process where only the annotation step is performed by the annotation team while data distribution and aggregation can only be done by a team supervisor.

Next, we describe the four steps of the process in detail. Later, we will describe how the individual steps of the process have been implemented in our tool CSniper, starting with Section 6.1.3.4.

Query

The annotator starts by running a query over the corpus to locate occurrences of the linguistic phenomenon in question. In order to locate relevant passages, the linguistic intuition of the annotators plays an important role. This intuition helps them to design a query that characterizes the phenomenon they are looking for. The query represents an initial hypothesis about the linguistic structures expected in these passages. It can be refined up to a point where the annotator observes a subjectively sufficient number of the phenomenon in the results, before beginning with the actual annotation.

The query may be substituted by an alternative mechanism which extracts potential occurrences of the phenomenon from the corpus. For example, machine learning may be used to find new passages which are very similar to passages that are known to contain the phenomenon.

Annotation

When a query yields good results, the annotator begins with the actual annotation task. Most likely, not all the query results are actually occurrences of the desired phenomenon. If that were the case, a simple rule-based annotation process would have been sufficient. On the contrary, results may require a detailed inspection of the surrounding document context in order to determine if they are an occurrence of the phenomenon in question or not.

Monitoring

The monitoring of annotation and query quality is an integral part of the process. There are two measures that can be monitored:

1. *Query quality* – The monitoring of the query quality is, to our knowledge, unique to the *annotation-by-query* process. After a user has annotated several results of a query, the precision of the query can be estimated, assuming that the occurrences of the phenomenon are distributed equally within the query results. Individual annotators can use this information to refine their queries by intellectually comparing the structure of their queries to synthesize a new one. The annotation team can use it to discuss promising querying strategies. Additionally, a systematic review of the queries used by the team can help the annotation team to come up with ideas for new queries that may cover variants of the desired phenomena not covered so far.

2. *Inter-annotator agreement* – Assessing the inter-annotator agreement is the second aspect of monitoring. While the overall inter-annotator agreement can be used to assess the general quality of annotations, looking in detail at disagreeing annotations helps to find particularly difficult examples that can be used to improve the annotation guidelines. This aspect is not specific to the *annotation-by-query* process, but is rather part of a distributed annotation process in general.

Aggregation

As every annotator in a distributed annotation process creates their own results, these eventually need to be aggregated into a final result. This aggregation can be done manually or automatically. As the *query-by-annotation* process is intended to be fully distributed, we opt for an automatic aggregation process. The aggregation strategies can range from a simple majority vote to manual curation.

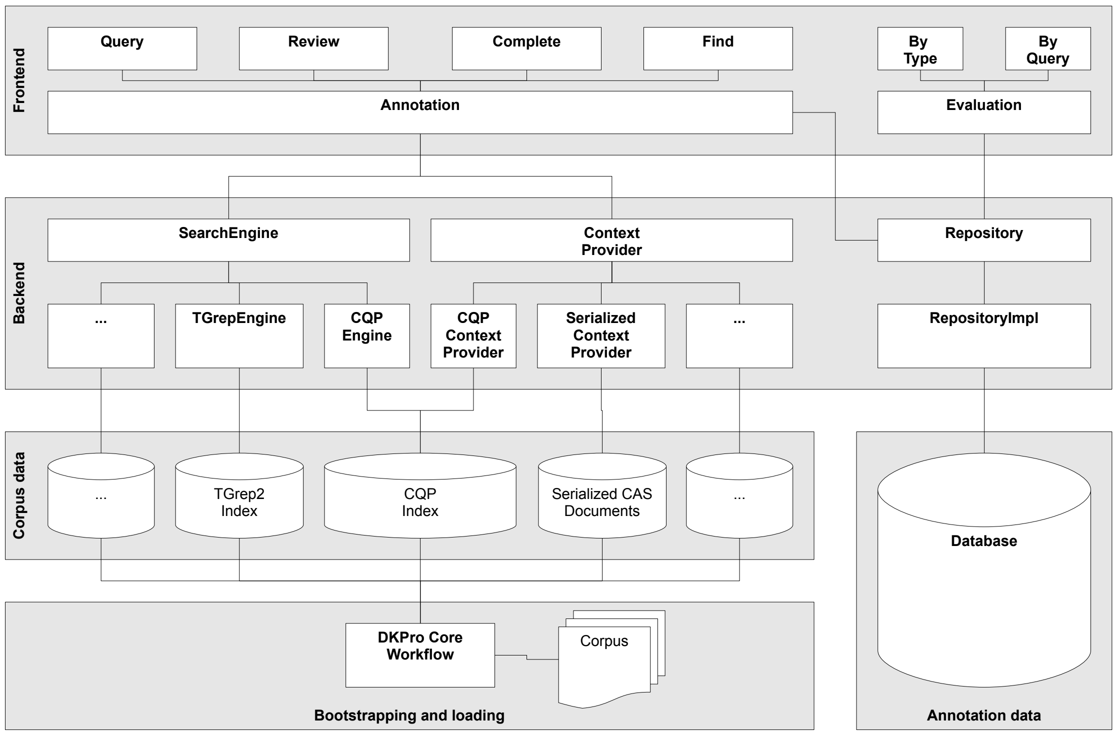


Figure 6.9: CSniper architecture overview

6.1.3.2 Architecture

To support the *annotation-by-query* process, we implemented a web-based tool called *CSniper* [50; 75]. Its overall architecture is divided into five parts (Figure 6.9 on page 162): *fronted*, *backend*, *corpus data*, *annotation data*, and *bootstrapping*.

Frontend

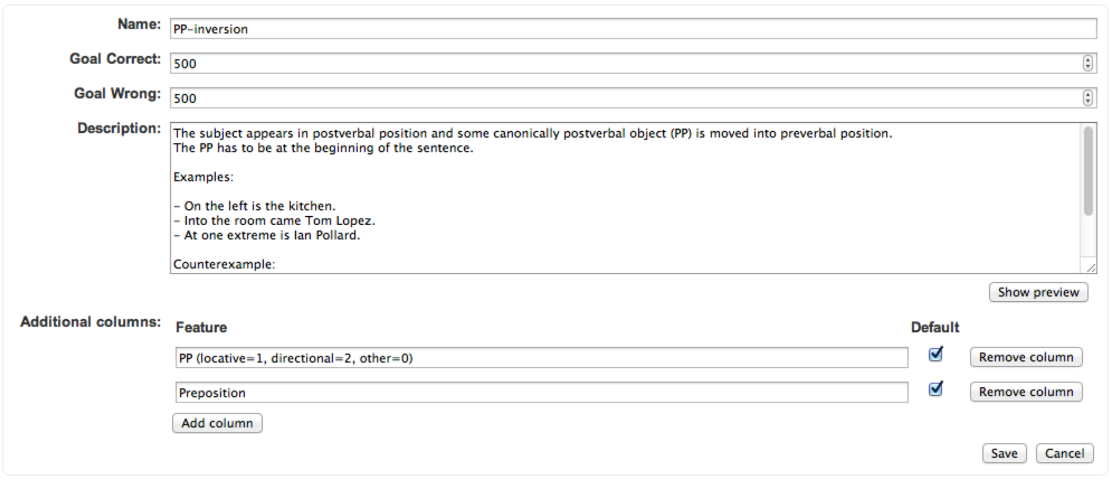
The browser-based frontend provides the user interface and user actions. Before annotating, the types of phenomena can be defined along with goals, i.e. how many occurrences of the phenomena the annotation team aims to locate. For annotation, actions include searching for phenomena, annotating search results, and reviewing annotations. It is also possible to monitor the state of annotations and evaluate inter-annotator agreement for certain phenomena or to examine the quality of queries. Auxilliary features, such as user management, are not covered in this thesis.

Backend

The frontend interacts with pluggable search engines and context providers, which provide access to the annotated corpora. Search engine plug-ins allow running a query over a corpus using a specific search engine. A context provider plug-in allows getting context for a search result, e.g. the preceding and the following text, optionally with annotations, which the annotator may need to decide whether a result is a true occurrence of the desired phenomenon.

Corpus data

Annotated corpora used by the tool need to be prepared in a set of different formats. Each search engine plug-in typically uses its own index format. E.g. we provide search plug-ins for TGrep2 [187] and for the IMS Open Corpus Workbench [77], which use their own index formats each. Context provider plug-ins have different requirements, e.g. the requirement to


Name: PP-inversion

Goal Correct: 500

Goal Wrong: 500

Description: The subject appears in postverbal position and some canonically postverbal object (PP) is moved into preverbal position.
The PP has to be at the beginning of the sentence.

Examples:

- On the left is the kitchen.
- Into the room came Tom Lopez.
- At one extreme is Ian Pollard.

Counterexample:

Additional columns:

Feature	Default
PP (locative=1, directional=2, other=0)	✓
Preposition	✓

Figure 6.10: CSniper type definition form

retrieve corpus data with their original line breaks and spacing, or the requirement to display annotations which are not included in the search indexes. Thus, they may again use their own data format. We include two context providers, one using the index of the IMS Open Corpus Workbench search plug-in and another one using serialized UIMA CASes which include the full annotation information.

Bootstrapping and loading

The search plug-ins used by CSniper require certain metadata to be encoded in the search indexes. This includes a collection ID, a document ID, and the offsets of words and sentences within the respective documents. This information is necessary for the context provider plug-ins to locate the context of a search result. The collection ID, document ID, and offsets are also used to anchor annotations created in CSniper to the corpus data.

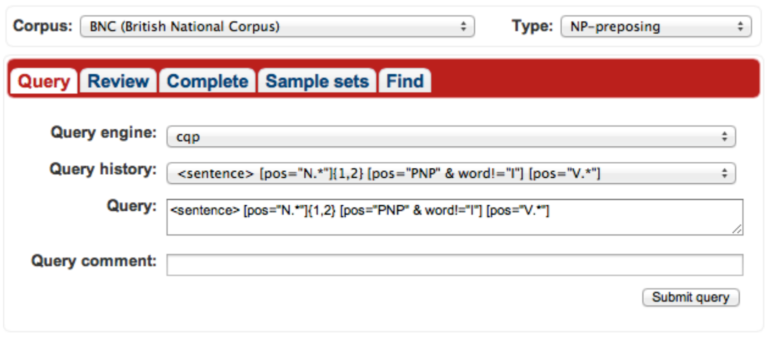
The DKPro Core collection (Section 5.2) includes components capable of writing indexes for the IMS Open Corpus Workbench and for TGrep2 containing the metadata that CSniper requires. It also includes a component to write the serialized UIMA CASes. This permits users to use analysis components integrated into DKPro Core to preprocess corpora before using them with CSniper. Alternatively, readily annotated corpora in various formats are directly supported by DKPro Core and can easily be converted. Such preprocessing or conversion workflows can be easily created and shared with other researchers as portable, executable workflows as suggested in Section 4.1.

Annotation data

In the current implementation of CSniper, the corpus data is read-only data, including any pre-created annotations. In particular, the currently used search engines do not support updating their indexes on-the-fly. For this reason, and in order to better support the monitoring functionality, all annotations made by the users are stored separately in a relational database.

6.1.3.3 Setting up types

Before the annotation can start, a set of interesting types of phenomena must be configured (Figure 6.10). The type definition consists of the name of the phenomenon and a guideline for the annotators. The guideline explains the phenomenon, how it can be identified, and possibly some examples. It is always visible on screen during the annotation process. A simple markup



Corpus: BNC (British National Corpus)
Type: NP-preposing
Query engine: cqp
Query history: <sentence> [pos="N.*"]{1,2} [pos="PNP" & word!="I"] [pos="V.*"]
Query: <sentence> [pos="N.*"]{1,2} [pos="PNP" & word!="I"] [pos="V.*"]
Query comment:

Figure 6.11: CSniper query form

language [127], as it is used in wiki webs, is supported to visually style the guideline. It supports features such as marking headings, italic text, bold text, or pre-formatted text.

It is possible to define goals, e.g. how many occurrences of the phenomenon the annotation team aims to find. The goals are used to measure the progress of the annotation team.

By default, annotators can only make default annotations: mark a search result as *correct* or *wrong* depending on whether it contains the desired phenomenon and leave a personal comment. However, it is possible to define arbitrary extra features which show up as additional columns in the annotation user interface.

6.1.3.4 Annotation

Next, we describe the user interface in CSniper used to perform the annotations. As mandated by the *annotation-by-query* process, this includes functionality for performing queries and annotations.

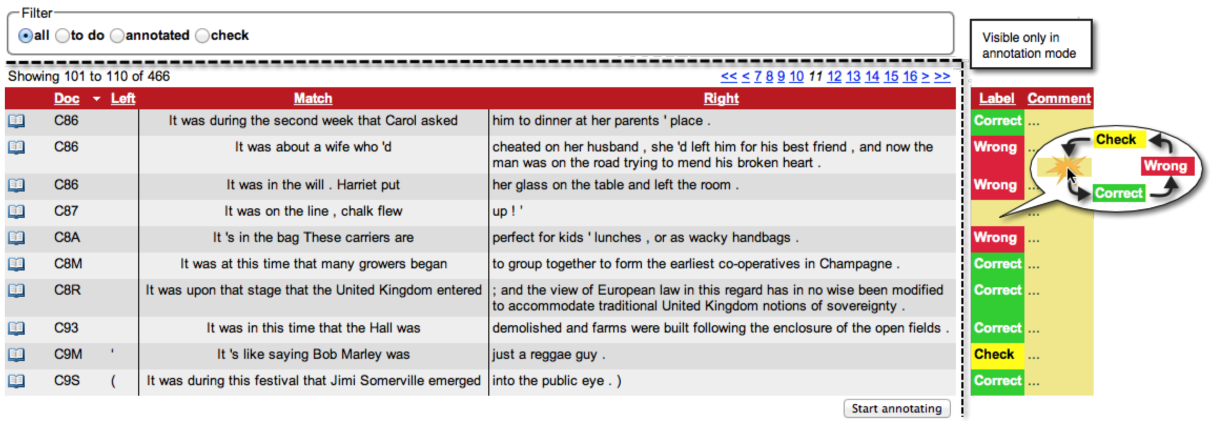
The user interface for performing queries has an intentionally simplistic design (Figure 6.11). Before any further action can be performed, the user must choose a corpus to work on and the type of phenomenon under examination. After that selection has been made, additional actions appear: *query*, *review*, *complete*, and *find*. Each of these actions retrieve results which can be annotated. We will first explain querying and annotation, before addressing the other actions.

Query

CSniper does not provide an abstraction over the query languages used by different search engine plug-ins. The query is passed on verbatim or with only minor additions to the underlying search engine. Thus, a search engine needs to be selected before a query can be entered. Passing the query directly to the search engine has several benefits. Users already familiar with the underlying search engines and their query languages do not need to learn an additional query language. Also, all features of the individual query languages can be used. New search engines can be integrated easily. However, there are also drawbacks. E.g. it is currently not possible to query multiple engines at the same time, e.g. in order to combine a query over the syntactic structure in TGrep2 with a query over lemmata in the IMS Open Corpus Workbench. If the query capabilities of individual search engines separately are not sufficient, it may be the easiest to implement a new plug-in to support a search engine with a more powerful query language, e.g. ANNIS.

Annotation

The query results are sentences which are displayed as a table. If the search engine supports identifying which part of the sentence matched the query statement, then a keyword-in-context presentation is used instead of a simple table (Figure 6.12 on page 165).


Filter
[✓ all] to do annotated check
Visible only in annotation mode
Check
Wrong
Correct
Start annotating

Showing 101 to 110 of 466   << < 7 8 9 10 11 12 13 14 15 16 > >>

	Doc	Left	Match	Right	Label	Comment
	C86		It was during the second week that Carol asked	him to dinner at her parents ' place .	Correct	...
	C86		It was about a wife who 'd	cheated on her husband , she 'd left him for his best friend , and now the man was on the road trying to mend his broken heart .	Wrong	...
	C86		It was in the will . Harriet put	her glass on the table and left the room .	Wrong	...
	C87		It was on the line , chalk flew	up ! '		...
	C8A		It 's in the bag These carriers are	perfect for kids ' lunches , or as wacky handbags .	Wrong	...
	C8M		It was at this time that many growers began	to group together to form the earliest co-operatives in Champagne .	Correct	...
	C8R		It was upon that stage that the United Kingdom entered	; and the view of European law in this regard has in no wise been modified to accommodate traditional United Kingdom notions of sovereignty .	Correct	...
	C93		It was in this time that the Hall was	demolished and farms were built following the enclosure of the open fields .	Correct	...
	C9M	'	It 's like saying Bob Marley was	just a reggae guy .	Check	...
	C9S	(	It was during this festival that Jimi Somerville emerged	into the public eye .)	Correct	...

Figure 6.12: CSniper query result and annotation interface (data: BNC [28])

If the query results do not match the desired phenomenon to the satisfaction of the user, the query can be refined. To help the user refine the query, it is possible to display the parse tree for each query result and the context of the result, including part-of-speech tags.

If the results appear satisfactory, the user can start annotating. Each query result, i.e. each sentence, can be annotated with one of the following labels:

- **Correct** – the search result matches the desired phenomenon.
- **Wrong** – the search result does not match the desired phenomenon.
- **Check** – the annotator is unsure and wants to get back to this result at a later time.
- **No label** – the search result is not annotated.

The annotation is performed by simply clicking into the *label* column in the row of the respective result (Figure 6.12). Clicking multiple times iterates through the different labels. If additional features have been defined for the selected type (cf. Section 6.1.3.3), these are also shown as additional columns and can be edited. In any case, each user can only see and edit their own annotations.

Reviewing annotations

Generally, there is no perfect agreement between annotators in any annotation task. There can be many reasons for this, e.g. true ambiguity, misinterpretation of the data, unclear annotation guidelines, or simply the annotator getting tired after a while. In any case, it is helpful for the annotation team to review their annotations from time to time, in particular those that the annotators do not agree upon.

The *review* mode allows each user examine and change their own annotations without performing a query. It is also possible to look specifically at *disputed* results on which the annotators do not agree. When examining disputed results, they also include results the user has not yet annotated, but which other members of the team have annotated and do not agree upon.

This mode is particularly useful when the annotators set up a meeting and can interactively discuss individual results and their annotations. The disputed results yield good new examples to be added to the annotation guidelines.

Ensuring multiple annotations per result

Every annotator is supposed to use his or her own linguistic intuition to create queries for those phenomena they are interested in. These queries are not shared between the annotators. In the worst case, every annotator may work on completely different search results. In the worst case, inter-annotator agreement would not be measurable, because every result has been annotated only by a single person.

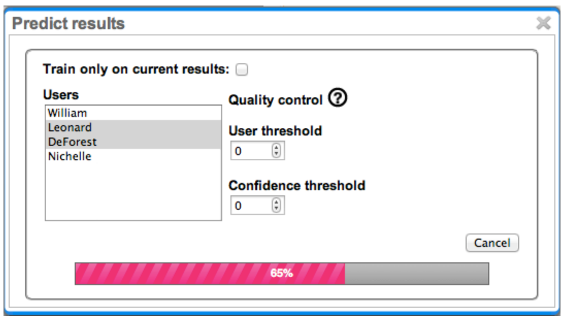
The analysis tool needs to implement a concept to avoid this situation. In CSniper, this is realized by an annotation mode called *complete*. In this mode, the user does not enter a query, but is presented with all search results that any fellow annotator has already annotated, but the current user has not. The annotation team should define regular intervals in which this function is used to make sure that all results have been annotated by all team members.

The current implementation of this concept could be further improved to allow the annotation team to reach their annotation goal faster. Consider the goal of finding 1,000 occurrences of the desired phenomenon. After some time, the annotators have repeatedly marked many of the search results as *wrong* and only a few have been marked repeatedly as *correct*. In order to meet the goal more rapidly, the *complete* feature could preferably present results to the user which already have been annotated as *correct* by fellow annotators.

Focusing on relevant results

While following the *annotation-by-query* process and using the tool CSniper, we observed that after some time, the annotation team had annotated those results which could be found by queries with a high precision. In order to find additional occurrences of the phenomena, it was necessary to relax the queries, which in turn led to many more results which needed to be inspected – most of them irrelevant.

To address this issue, we introduced an annotation assistant functionality called *predict*. It trains a classifier on already annotated results for the phenomenon in question (cf. Section 6.1.3.4) and uses that to classify the results of the query the user is currently working on as *predicted correct* or *predicted wrong*. By sorting the results according to their predicted label, the user can focus on more relevant results when a query is relaxed.


Predict results

Train only on current results: ☐

Users: William, Leonard, DeForest, Nichelle

Quality control

User threshold: 0

Confidence threshold: 0

Figure 6.13: CSniper annotation suggestion configuration dialog

Because the user can select which members of the annotation team the classifier should be trained on, this functionality also provides a way for users to indirectly interact with each other without being able to see the annotations of the other annotators. We believe that an annotator objectively judges the suggestions from the classifier and is less likely to be biased by its suggestions. The ability to see annotations from trusted fellow annotators may cause a stronger bias.

The data on which the classifier is trained can be configured to include only annotations from certain members of the annotation team and to include only annotations that meet a certain quality (Figure 6.13). The quality control settings are described in detail in the next section (Section 6.1.3.5).

The machine learning functionality currently integrated in CSniper is training a tree kernel support vector machine on constituency parse trees using *SVM-LIGHT-TK* [162; 124]. The constituency structure, the part-of-speech tags, and the words themselves are the only features currently being taken into account.

Finding results automatically

The occurrences of a phenomenon that can be found by a query are limited by the intuition of the query writer. For this reason, we implemented a second annotation assistant functionality called *find* using the same classifier training mechanism as before. The *predict* feature described in the previous section was applied only to the results of the last search. However, the *find* feature allows the user to train a classifier based on the already annotated results for the phenomenon under examination and then applies this classifier to *all* sentences in the corpus. This feature aims to exploit the ability of the classifier to generalize over the training data in order to find new occurrences of the phenomenon that have not been covered by queries yet.

Because a corpus can be large and actually classifying all sentences may take too much time in the current implementation, the corpus is partitioned into batches of 1,000 sentences each, which are then randomly selected and classified. The process terminates when 1,000 positively classified results have been found. As even this may take a long time, the process can be interrupted at any time, delivering all positively classified results up to this point.

6.1.3.5 Monitoring

The monitoring functionality provides the annotation team with information about the state of their annotation project. Progress and quality can be observed from two vantage points: based on the annotated results and based on the queries. In both cases, the quality measures depend on how the annotations from different users are aggregated into the result which is evaluated. This aggregation happens on demand and its results are not persisted. However, a user can export the aggregated results to a file, e.g. to further evaluate them in another tool.

Aggregation

CSniper uses a configurable automatic aggregation which does not require the annotation team to perform an explicit curation step. The same aggregation mechanism is also used when generating the data for training the classifiers mentioned previously.

The aggregation process is controlled using two parameters set by the user, e.g. when monitoring results (Figure 6.14 on page 168) or when configuring the automatic prediction of labels (Figure 6.13 on page 166).[1]

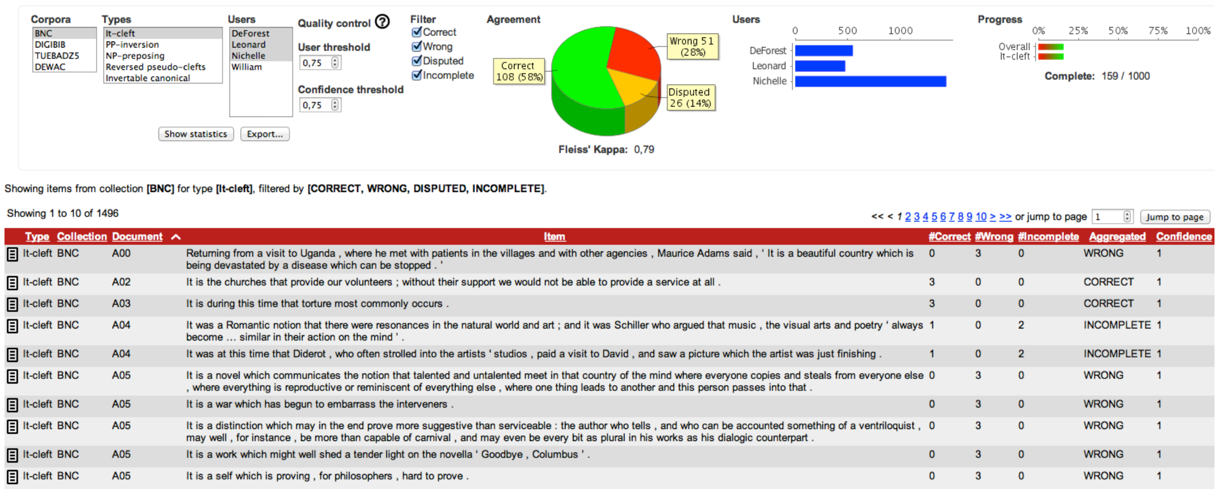
- **Participation threshold** – the proportion of members of the annotation team who are required to annotate a result before it is considered to be annotated. With a threshold of 0, all results that have been annotated by at least one team member are taken into account. With a threshold of 1, a result must have been annotated by all team members, otherwise it counts as *incomplete*. The participation for a result is calculated by dividing the number of *correct* and *wrong* labels by the annotation team size:

$$participation() = \frac{correct + wrong}{teamsize} \tag{6.1}$$

- **Confidence threshold** – the extent by which the majority has to win over the minority. A confidence threshold of 0 results in a simple majority vote and a result is only considered disputed if there is a draw. A threshold of 1 requires all annotators to agree, otherwise the result is considered *disputed*. The confidence towards a result being *correct* or *wrong* is calculated as follows:

$$confidence(label) = \frac{label}{max(correct, wrong)} \qquad label \in \{correct, wrong\} \tag{6.2}$$

[1] Note that the participation threshold is shown as *user threshold* in these figures.



Corpora: BNC, DIGIBIB, TUEBADZ5, DEWAC

Types: It-cleft, PP-inversion, NP-preposing, Reversed pseudo-clefts, Invertable canonical

Users: DeForest, Leonard, Nichelle, William

Quality control

User threshold: 0,75

Confidence threshold: 0,75

Filter: ☑ Correct ☑ Wrong ☑ Disputed ☑ Incomplete

Fleiss' Kappa: 0,79

Complete: 159 / 1000

Show statistics Export...

Showing items from collection **[BNC]** for type **[It-cleft]**, filtered by **[CORRECT, WRONG, DISPUTED, INCOMPLETE]**.

Showing 1 to 10 of 1496

<< < 1 2 3 4 5 6 7 8 9 10 > >> or jump to page 1 Jump to page

Type	Collection	Document	Item	#Correct	#Wrong	#Incomplete	Aggregated	Confidence
It-cleft	BNC	A00	Returning from a visit to Uganda , where he met with patients in the villages and with other agencies , Maurice Adams said , ' It is a beautiful country which is being devastated by a disease which can be stopped . '	0	3	0	WRONG	1
It-cleft	BNC	A02	It is the churches that provide our volunteers ; without their support we would not be able to provide a service at all .	3	0	0	CORRECT	1
It-cleft	BNC	A03	It is during this time that torture most commonly occurs .	3	0	0	CORRECT	1
It-cleft	BNC	A04	It was a Romantic notion that there were resonances in the natural world and art ; and it was Schiller who argued that music , the visual arts and poetry ' always become … similar in their action on the mind ' .	1	0	2	INCOMPLETE	1
It-cleft	BNC	A04	It was at this time that Diderot , who often strolled into the artists ' studios , paid a visit to David , and saw a picture which the artist was just finishing .	1	0	2	INCOMPLETE	1
It-cleft	BNC	A05	It is a novel which communicates the notion that talented and untalented meet in that country of the mind where everyone copies and steals from everyone else , where everything is reproductive or reminiscent of everything else , where one thing leads to another and this person passes into that .	0	3	0	WRONG	1
It-cleft	BNC	A05	It is a war which has begun to embarrass the interveners .	0	3	0	WRONG	1
It-cleft	BNC	A05	It is a distinction which may in the end prove more suggestive than serviceable : the author who tells , and who can be accounted something of a ventriloquist , may well , for instance , be more than capable of carnival , and may even be every bit as plural in his works as his dialogic counterpart .	0	3	0	WRONG	1
It-cleft	BNC	A05	It is a work which might well shed a tender light on the novella ' Goodbye , Columbus ' .	0	3	0	WRONG	1
It-cleft	BNC	A05	It is a self which is proving , for philosophers , hard to prove .	0	3	0	WRONG	1

Figure 6.14: CSniper inter-annotator agreement monitoring (data: BNC [28])

The aggregation process assigns one of the following labels to each search result:

- **Correct** – the search result matches the desired phenomenon.
- **Wrong** – the search result does not match the desired phenomenon.
- **Disputed** – a sufficient number of annotators have annotated the search result (as controlled by the *participation threshold*), but the difference between the number of *correct* and *wrong* labels assigned by the annotators is not above the specified *confidence threshold*.
- **Incomplete** – at least one annotator has annotated the search result, but the proportion of team members who have annotated the search result is below the *participation threshold*.

Result-based monitoring

CSniper provides a detailed account over the state of annotations (Figure 6.14). After selecting a corpus, a type (cf. Section 6.1.3.3), and the users whose annotations to take into account, a tabular overview over all matching annotated results is provided. This includes how often each result was annotated as *correct* and *wrong*, how many of the selected users have not yet annotated the result, the aggregated result according to the current quality control settings, and the confidence.

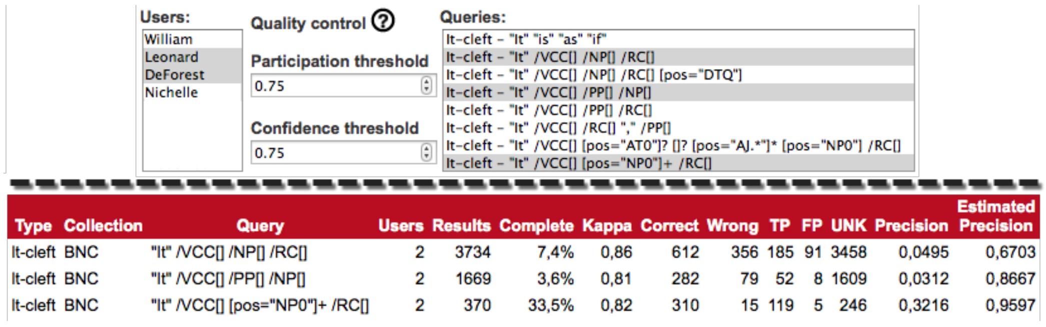
A pie-chart provides a visual indication of the current agreement and shows the total number of *correct*, *wrong*, and *disputed* results after aggregation. Note that we did not include the *incomplete* results in this chart. It is easy for a single user to generate new incomplete results by annotating results that have not been annotated before. This led to some users finding it frustrating to see the number of incomplete items grow all the time. To avoid this, we introduced the ability to define goals for each type and display the progress towards the goals (*Progress* chart in Figure 6.14). Progress towards the goals is calculated based on the specified values for *participation threshold* and *confidence threshold*.

E.g. if the goals have been set to 500 *correct* and 500 *wrong* results for a specific type, and if – based on the thresholds – the results from the annotation team are aggregated into 250 *correct*, 700 *wrong*, 100 *disputed*, and 50 *incomplete* results, then the progress for the type is 75%:

$$progress() = \frac{min(correct, goal_{correct}) + min(wrong, goal_{wrong})}{goal_{correct} + goal_{wrong}} \tag{6.3}$$

Query-based monitoring

It is also possible to examine the state and progress of annotations based on the queries (Figure 6.15). Progress for queries is tracked relative to the number of results that a query produces. Again, quality control settings based on the participation threshold and the confidence threshold are used to define how the aggregation of annotations by different users is performed. The precision of a query is calculated from the results labeled as *correct* by the aggregation process. However, if not all results of the query have been annotated yet, this number is misleadingly low. Assuming an equal distribution of *correct* results across the whole query, an additional *estimated precision* is provided, which aims to better approximate the true precision of the query.



Users:
William
Leonard
DeForest
Nichelle

Quality control

Participation threshold: 0.75

Confidence threshold: 0.75

Queries:
It-cleft – "It" "is" "as" "if"
It-cleft – "It" /VCC[] /NP[] /RC[]
It-cleft – "It" /VCC[] /NP[] /RC[] [pos="DTQ"]
It-cleft – "It" /VCC[] /PP[] /NP[]
It-cleft – "It" /VCC[] /PP[] /RC[]
It-cleft – "It" /VCC[] /RC[] "," /PP[]
It-cleft – "It" /VCC[] [pos="AT0"]? []? [pos="AJ.*"]* [pos="NP0"] /RC[]
It-cleft – "It" /VCC[] [pos="NP0"]+ /RC[]

Type	Collection	Query	Users	Results	Complete	Kappa	Correct	Wrong	TP	FP	UNK	Precision	Estimated Precision
It-cleft	BNC	"It" /VCC[] /NP[] /RC[]	2	3734	7,4%	0,86	612	356	185	91	3458	0,0495	0,6703
It-cleft	BNC	"It" /VCC[] /PP[] /NP[]	2	1669	3,6%	0,81	282	79	52	8	1609	0,0312	0,8667
It-cleft	BNC	"It" /VCC[] [pos="NP0"]+ /RC[]	2	370	33,5%	0,82	310	15	119	5	246	0,3216	0,9597

Figure 6.15: CSniper query quality monitoring

Additionally, it is possible via the query-based monitoring to see queries that other team members have come up with. These can serve as a starting point or inspiration for new queries, e.g. in order to increase precision or to find additional occurrences of a phenomenon which are not covered by existing queries.

6.1.4 Identification of non-canonical constructions

The *annotation-by-query* approach and the *CSniper* tool were used by linguists in a research project for the identification of non-canonical constructions. A non-canonical construction (NCC) deviates from the canonical structure predominantly used in a language, e.g. the *subject-verb-object* order in English. The use-case given is a summary of what has been described by Eckart de Castilho et al. [75] and Radó [182].

Some examples for such NCCs are given in Figure 6.16. Even though these examples all express the same fact, the variation in the ordering of information puts certain information into the focus and yields different effects on the levels of discourse and pragmatics.

1. *[S: The media] was now [V: calling] [O1: Reagan] [O2: the frontrunner].* *(canonical)*
2. *It was [O1: Reagan] whom [S: the media] was now [V: calling] [O2: the frontrunner].* *(it-cleft)*
3. *It was [S: the media] who was now [V: calling] [O1: Reagan] [O2: the frontrunner].* *(it-cleft)*
4. *It was now that the [S: the media] were [V: calling] [O1: Reagan] [O2: the frontrunner].* *(it-cleft)*
5. *[O1: Reagan] [S: the media] was now [V: calling] [O1: Reagan] [O2: the frontrunner].* *(inversion)*

Figure 6.16: Example of a canonical sentence and several non-canonical variations (cf. [75])

The different kinds of NCCs appear with a relatively low frequency in corpora, some more frequent than others. For this reason, a large corpus is required in order to perform a corpus-linguistic study of these phenomena.

Additionally, the phenomena are ambiguous on the surface. For example, an *it-cleft* construction needs to be distinguished from an anaphoric *it* or from constructions using true relative clauses (cf. Figure 6.17). The annotator requires access to the larger context in order to disambiguate such cases.

6. *London will be the only capital city in Europe where rail services are expected to make a profit,' he added. It is a policy that could lead to economic and environmental chaos.* *(anaphoric) [BNC: A9N-s400]*
7. *It is a legal manoeuvre that declined in currency in the '80s.* *(relative clause) [BNC: B1L-s576]*

Figure 6.17: Example of surface ambiguity of *it-cleft* constructions (cf. [75])

Bootstrapping workflow

For the corpus linguistic research on these NCCs, we used the *British National Corpus* (*BNC*) [28] and the *TüBa-D/Z* [216]. These corpora are already pre-annotated for several linguistic features, such as segmentation, part-of-speech tags, lemma. TüBa D/Z also includes syntactic constituency information, while this information is not part of the BNC.

Using DKPro Core, we converted these corpora to the index and document formats used by CSniper. Where necessary, more annotations were added, to offer the same depth of analysis on all corpora. In particular, we parsed the BNC with the Stanford parser using a factored parser model [131]. We used DKPro Core (Section 5.2) and DKPro BigData [61] to set up an analysis workflow with this parser and to run it on an Apache Hadoop [8] cluster.

Analysis workflow

Using these pre-annotated corpora, the annotators were able to express their linguistic knowledge and intuition about the NCCs they were looking for as queries. The annotation team started with queries based on part-of-speech and lemma patterns using the IMS Open Corpus Workbench engine. While these queries worked very well for some NCCs, e.g. *It-cleft* sentences, they were not well suited for others, e.g. *NP-preposing* (Figure 6.18). These could be better captured by queries over the constituency structure using the TGrep2 engine.

NP-preposing:
A noun-phrase object of the verb appears before the subject in sentence-initial position.

8. *A bagel I can give you.* *(NP-preposing)*
9. *Basketball, I like a lot better.* *(NP-preposing)*

Figure 6.18: Examples of *NP-preposing* from the annotation guidelines used by Eckart de Castilho et al. [75]

The granularity level for search results is a sentence. The corpora already contained an annotation of sentence boundaries. We kept the annotation process simple by asking that users only annotate if a search result contains the NCC in question. We did not ask where exactly the NCC is located within the sentence. This binary annotation was sufficient for the annotators to quickly process the results and the linguists to perform their study (cf. [182]).

Reception

In summary, the *annotation-by-query* process and the CSniper tool were well-received by the annotation team of at times up to six people, consisting of linguists, computational linguists, and computer scientists. Although the approach does not find all occurrences of a particular phenomenon, it allows the annotation team to find many occurrences with a relatively low effort. The embedded machine learning functionality further helped to find additional occurrences which were not covered by the queries.

The requirements for special interactions supporting the *annotation-by-query* process, such as the ability to review and the support for ensuring multiple annotations per result, were direct consequences of applying the process in the distributed team. Also, the ability of adding arbitrary custom columns to the annotation user interface was a user requirement. It was used to further classify the sentences in which certain NCCs appeared. E.g. in case of PP-inversion[2], it was recorded whether the first constituent of the sentence was a pronoun or not, whether it was anaphoric or elliptic, whether the PP was locative or directional, etc.

6.1.5 Summary

We have presented the *annotation-by-query* approach of querying a corpus for sentences matching linguistic patterns, annotating them, monitoring the progress and quality of the annotations, and automatically aggregating annotations from multiple annotators based on quality thresholds. The approach is well-suited for annotation tasks that require manual analysis over large corpora to locate ambiguous phenomena on the basis of queries and subsequently annotate these manually by a team of annotators.

In order to use the *annotation-by-query* approach, pre-annotated corpora are required, based on which linguistic queries can be formulated. This highlights the aspect of interaction between automatic and manual annotation in this process, as these corpora can be created using automatic analysis workflows. In the present case, we used workflows based on DKPro Core (Section 5.2), which read existing annotated corpora, augmented the annotations, and produced the index formats used by the search engines underlying CSniper. Of course, these indexes can be used without CSniper, directly with the respective tools, e.g. the IMS Corpus Workbench or TGrep2.

A further opportunity for the integration of automatic analysis with the manual annotation process was taken by using a machine learning approach to help the human annotators focus on particularly relevant results in cases that queries yield too many irrelevant results. This also augments the distributed working mode itself, because it allows the users to indirectly interact with each other.

A unique feature of our approach is that all members of the annotation team are equal. There is no administrator or curator role with additional privileges, such as distributing work or aggregating results. This works well for the binary *correct/wrong* annotations that we needed in our task. Future research might examine how to extend this approach to more complex kinds of annotations, and how such an approach could be used to set up an open annotation platform to host distributed annotation projects for e.g. constituency parses, dependency parses, coreference analysis, etc. For example, consider an online annotation platform which contains a certain corpus. Many groups of researchers working on that corpus could create and share their annotations and their guidelines there. Such a platform would provide a completely new way for language researchers to interact and cooperate with each other.

[2] From the guidelines used in the project: *The subject appears in postverbal position and some canonically postverbal object (PP) is moved into preverbal position. The PP has to be at the beginning of the sentence. Example: "On the left is the kitchen".*

7 Conclusion

In this thesis, we have studied the integration of automatic analysis of text with manual analysis. Due to the current trend of combining natural language analysis with questions from the humanities, this is an important topic. In collaboration with researchers from the humanities, we developed a scenario which illustrates the cooperation of a linguist, a computer scientist, and a computing center operator (Figure 7.1). This scenario and the issues that it highlights are prototypical of collaborations between humanities and computer science researchers.

We have grouped the issues discovered in this scenario by four general principles: *reproducibility*, *usability*, *flexibility*, and *interactivity*. To facilitate the integration of automatic and manual analysis, we have addressed all these issues, while observing that improving support for one of these principles does not come at the expense of another one. We now summarize our contributions towards each of these principles in turn.

Reproducibility

We summarize our contributions towards reproducibility under two aspects: *portability* which promotes the sharing of analysis workflows between researchers, and *automatization* which removes error prone manual steps from experimental setups.

Portability

An important aspect in our scenario is the assembly and exchange of automatic analysis workflows between the different parties, in order to reproduce results or to apply the analysis workflows to new data.

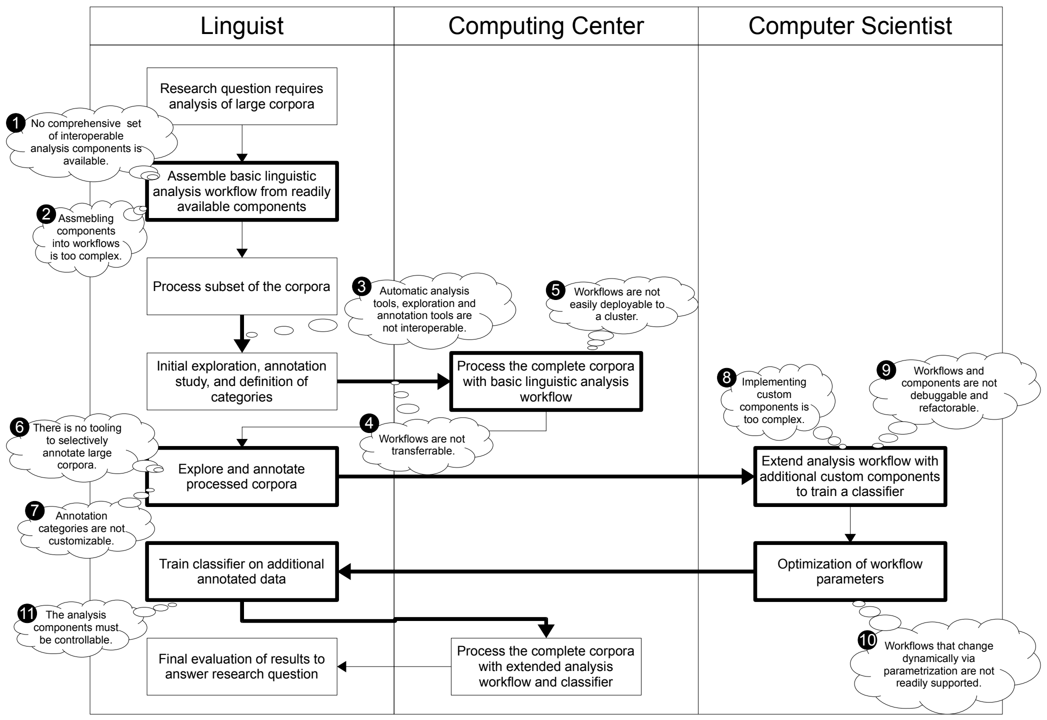

Linguist
Computing Center
Computer Scientist
Research question requires analysis of large corpora
1 No comprehensive set of interoperable analysis components is available.
Assemble basic linguistic analysis workflow from readily available components
2 Assmebling components into workflows is too complex.
Process subset of the corpora
3 Automatic analysis tools, exploration and annotation tools are not interoperable.
5 Workflows are not easily deployable to a cluster.
Initial exploration, annotation study, and definition of categories
Process the complete corpora with basic linguistic analysis workflow
8 Implementing custom components is too complex.
9 Workflows and components are not debuggable and refactorable.
4 Workflows are not transferrable.
6 There is no tooling to selectively annotate large corpora.
Explore and annotate processed corpora
Extend analysis workflow with additional custom components to train a classifier
7 Annotation categories are not customizable.
Train classifier on additional annotated data
Optimization of workflow parameters
11 The analysis components must be controllable.
10 Workflows that change dynamically via parametrization are not readily supported.
Final evaluation of results to answer research question
Process the complete corpora with extended analysis workflow and classifier

Figure 7.1: Use case: a linguist and a computer scientist collaborate on analyzing a large corpus

We have observed that current approaches to describing automatic analysis workflows do not contain sufficient information to take a workflow from one computer and run it on another computer. In particular, there are no versioned references to analysis components and resources. An approach is required which allows describing analysis workflows in a concise and self-contained manner.

We also noted that current approaches that aim to provide easily usable and reproducible analysis workflows tend to rely on third-party web services. This prevents the researcher from maintaining control over the experimental setup. The setup is subject to decay as these third-party services are upgraded or as they become unavailable. In order to maintain control, the experimental setup must rely on portable software and resources which allows each researcher to maintain their own copies.

Additionally, we have found that current approaches to packaging resources, e.g. part-of-speech tagger or parser models, and distributing them via repositories make it difficult to share the resources across different analysis tools, component collections, and processing frameworks. For example, often resources are packaged directly with the analysis components that require them, instead of being distributed separately.

Our use-case scenario (Figure 7.1 on page 173) lists the issues mentioned above as:

❹ Workflows are not portable between computers.
❺ Workflows are not easily deployable to a compute cluster.
⓫ The user has no control over workflows that rely on expert skills from a different domain, undocumented knowledge, or third-party infrastructures, e.g. web services.

To address these issues, we have described an approach to implementing analysis workflows in a self-contained manner based on portable analysis components and resources (Section 4.1). We have demonstrated this approach using a Groovy script which references all required analysis components and optionally all resources. Groovy is capable of automatically provisioning all dependencies from a repository, e.g. Maven Central [149], to the computer on which it is run. Such a script can be easily exchanged between the different parties in our scenario, allowing each one to run the analysis workflow. This facilitates reproducing the experimental results of other researchers, but it also facilitates applying the analysis workflow to new data.

As part of our approach, we have defined a best practice for packaging resources, so that only one copy of the resource needs to be maintained in a repository, but that copy can be used by different analysis tools, component collections, and processing frameworks (Section 3.1).

In addition to providing reproducibility, our approach provides usability because the user does not need to manually download and install tools and resources.

In future work, the assembly analysis workflows for non-expert programmers should be further facilitated. Writing a script requires prior knowledge about the available analysis components, where they can be found, and the parameters of each component. An interactive tool with a graphical user interface could allow the user to browse through the components and to assemble them into an analysis workflow. The workflow could then be saved in an executable form, e.g. as a Groovy script, so that it can be easily shared with other users.

Additionally, further research should examine how convenient solutions such as our approach for addressing, packaging, and distributing resources can be extended to primary data and can be combined with nascent metadata schemes like ISOcat [130] and the *Component Metadata Infrastructure* (*CMDI*) [34]. While these schemes offer more detailed metadata, repositories using them, e.g. the CLARIN Virtual Language Observatory [223], do not offer the same capabilities of automatically downloading and using resources, as our Maven-based approach provides.

Automatization

We have observed that current processing frameworks do not readily support parameter sweeping experiments in which analysis workflows are executed repeatedly with different sets of configuration parameters. They also do not support workflows that change their structure based on their parametrization. Additionally, they expect that a workflow can process the input data one document at a time and do not support intermediate aggregation steps, e.g. counting the number of all tokens in the data, and using the result of such an aggregation in further processing steps. Alternative workflow engines tend to set up completely new development environments, workflow description languages, and often target grid computing. This incurs additional complexity in learning, developing, and debugging such workflows.

Our use-case scenario (Figure 7.1 on page 173) lists the issues mentioned above as:

❾ Workflows and components are not sufficiently debuggable and refactorable.
❿ Workflows that change dynamically via parametrization are not readily supported.

We have described an approach to analysis workflows that allows them to change their structure dynamically based on their parametrization, e.g. in parameter sweeping experiments (Section 4.2). We follow a programmatic approach that allows computer scientists and other advanced users with programming skills to employ the debugging and refactoring capabilities of modern integrated development environments. Our approach allows modeling workflows based on tasks that need to be performed and on data dependencies between these tasks. These concepts are implemented in a lightweight manner, which makes it easy to take arbitrary existing analysis workflows and integrate them into a comprehensive experimental setup, thus removing the need to manually run individual workflows and to manually forward data from one workflow to the next. Additionally, our concept of a parameter space integrates the ability to perform parameter sweeping experiments. Dynamic workflows are enabled by the fact that our approach allows changing data dependencies between tasks based on the current parametrization.

In future work, we plan to investigate how to facilitate the deployment of workflows built with our approach on compute clusters. We plan to build on the recent DKPro BigData [61] project, in order to deploy workflows to a cluster based on Apache Hadoop [8].

Usability

We have found that the analysis workflow descriptions used by current processing frameworks tend to be long and verbose because many parameters have no sensible default values or no default values at all. In particular those parameters that instruct an analysis component to use a certain resource are typically mandatory and have no default value. An approach is required to allow analysis components to automatically decide which resources they require based on the data they are processing.

Also, we have noted that influencing the behavior of an analysis component using only simple parameters is not sufficient in certain kinds of analysis workflows. E.g. when setting up a machine learning workflow, the user should be able to conveniently configure the feature extraction without requiring special programming skills. Existing approaches to influence the behavior of components via parameters do not sufficiently cover such cases. Alternative approaches, e.g. using inheritance, require considerable programming skills.

Our use-case scenario (Figure 7.1 on page 173) lists the issues mentioned above as:

❷ Assembling automatic analysis components into workflows is too complex.
❽ Implementing new interoperable automatic analysis components is too complex.

We have described an approach that allows analysis components to address and acquire such resources automatically at runtime, based on the data that is being processed (Section 3.1). Our approach contributes to keeping analysis workflow descriptions concise, because it removes the need to define the resources to be used for each analysis component, unless the user explicitly wants to use a non-default resource. Also, the resources can be downloaded automatically at runtime, which removes the need of deciding in advance which resources can be used.

We have described an approach to configuring the behavior of analysis components using the strategy pattern of object-oriented programming (Section 3.2). This pattern is not sufficiently supported by existing processing frameworks at the level of configuring analysis components. We extended the uimaFIT library [14] to provide the necessary support for the UIMA framework [10]. Our approach provides a new level of flexibility when configuring analysis components.

Future work should introduce support for default strategies to further improve the usability of our approach. Just as default parameter values, these strategies are used if the user does not explicitly configure another strategy.

Flexibility

We have found that most current collections of analysis components tend to focus only on the analysis tools of a specific vendor, on a specific task, or they rely on third-party web-services, which we already considered as being problematic due to the user's lack of control over them. To provide the user with the flexibility of building analysis workflows for different tasks a component collection has to offer a rich choice of analysis components and resources for each of the supported analysis tasks, e.g. part-of-speech tagging or parsing.

We have noted that, while analysis components within a component collection are typically interoperable, they are not interoperable with components from other collections, because each collection uses its own annotation type system. Additionally, different tools tend to support certain annotation type system designs better than others, e.g. designs in which annotation type names represent linguistic categories, e.g. part-of-speech tags. An analysis of the type systems used by different component collections is required to better understand the different approaches to type system design and their specific benefits and drawbacks.

Our use-case scenario (Figure 7.1 on page 173) lists the issues mentioned above as:

❶ No comprehensive set of interoperable automatic analysis components is available.
❸ Automatic analysis tools and annotation editors are not interoperable.
❼ In automatic analysis, annotation type systems are predefined, but manual annotation requires customizability.

We have revised and extended DKPro Core (Section 5.2), our open-source collection of UIMA components. We have improved its usability, e.g. by including the automatic selection and acquisition of resources and packaged resources, and by introducing a set of common parameters across the component collection to ensure that users do not have to learn a completely new set of parameters every time they use a new component. While all our components are interoperable at the level of the type system, we have analyzed their interoperability at the conceptual level and their interchangeability within the workflow. We have found that most components that serve for the same analysis task, e.g. part-of-speech tagging or parsing, consume the same input and produce the same output and are therefore interchangeable within the workflow. At the conceptual level, interoperability depends on the tag sets and on the annotation guidelines. We have found that for some languages, e.g. English and German, there appears to be a good interoperability, because the models largely use compatible tag sets. For other languages, there are various tag sets in use, thus, they are not directly interoperable. We have observed that tag sets may appear to be largely similar, but even small differences are clues that they in fact use different underlying annotation guidelines. Therefore, we recommend that it should not only be possible to extract a list of tags from model files, but that annotation guidelines and tag sets should have identifiers and versions which should be extractable from the model files. Likewise, the provenance of any resources and analysis components used in an analysis workflow should be recorded as part of the analysis results, to make it easier for the user to give attribution to their original authors and providers when experimental results are published.

We have performed an analysis of the annotation type systems of different UIMA-based component collections (Section 5.1). It was our aim to discover different patterns of type system design and to determine the feasibility of consolidating parts of these type systems into a common type system. We have discovered several commonly used structural patterns, e.g. trees, linked lists, and sets, which deserve special support by annotation editors. We have discovered different strategies of associating annotations with each other, e.g. via co-indexing, or explicit linking, have discussed their benefits and drawbacks, and have given recommendations how to choose which strategy to use. We have discovered different approaches to modeling labels, have discussed their benefits and drawbacks, and have given recommendations how to choose which approach to use. We have discussed different approaches and motivations to organizing sets of annotations as annotation layers and have provided advice how to choose between the approaches. We have directly compared the representation of sentences and tokens in the different type systems and have concluded that a consolidation should happen bottom-up and starting from type systems that have few differences. Finally, we have found that the UIMA meta model currently is not able to express and test many constraints which are expected by analysis components, e.g. that token annotations must not cross sentence boundaries. We recommend that it should be possible to formulate such constraints, particularly when users create annotations manually, to ensure the manual annotations are compatible with the expectations of automatic analysis components.

In addition to offering flexibility, our component collection provides usability by integrating our approach for automatically selecting and installing resources. Since our components are packaged in a portable way and distributed via a repository infrastructure, they also contribute to reproducibility.

In future work, we plan to integrate more metadata into the DKPro Core component collection. In particular, we consider adding the ability to track the provenance of resources, analysis components, and results as an important aspect. We also plan to continually grow and update the component collection to further increase the coverage of different analysis tasks, languages, tag sets, and domains.

Future work should also continue to investigate options of consolidating different component collections and their type systems.

Interactivity

We have noted that the manual annotation of infrequent phenomena in large corpora requires a new kind of annotation tool. In particular, it is necessary that the corpus can be searched efficiently and that the results of this search can be annotated immediately. Existing annotation tools only have limited search capabilities. Existing linguistic search engines do not permit the user to make annotations. Additionally, an approach is required by which multiple annotators can search and annotate in parallel, because this allows determining the annotation quality in terms of inter-annotator agreement.

In our use-case scenario (Figure 7.1 on page 173), the issue mentioned above is listed as:

❻ There are no adequate tools for the selective annotation of large corpora.

We have described the *annotation-by-query* approach which combines searching, annotating, evaluating, and aggregating annotations into an integrated process (Section 6.1). Contrary to previous approaches, our process allows every step to be performed by every member of the annotation team. By integrating search and annotation, it becomes possible to interactively locate potential occurrences of the infrequent phenomenon in question and annotate these at the same time. We have implemented this process in the novel open-source annotation tool *CSniper* [75; 50]. Additionally, CSniper provides the annotator with machine-learning based suggestions, to help the annotator focus on relevant search results. The suggestions are generated by a support vector machine using a tree kernel which is trained on the constituency parse trees of already annotated search results. The set of annotated results used for training can be configured, e.g. based on the annotators who created them and on their agreement. To use a corpus with CSniper, it should already be annotated for sentences, tokens, part-of-speech tags, lemmata, and constituency parse trees. Corpora that do not offer these annotations can be automatically annotated with a workflow using DKPro Core components.

Our approach provides a way of interactively exploiting annotated corpora in search of infrequent phenomena. We benefit from our previously presented contributions towards usability, flexibility, and reproducibility. E.g. the self-contained portable analysis workflows and DKPro Core can be used to automatically preprocess corpora and convert them to the formats used by CSniper. CSniper uses DKPro Core and UIMA for internal storage and processing, e.g. when generating annotation suggestions.

As future work, it could be considered to extend the idea of an open annotation platform as it has been implemented in CSniper, in which all annotators are equals and in which annotations are aggregated automatically and on-demand. Many annotated corpora are subject to license restrictions preventing their free use and redistribution. An open annotation platform allowing interested members of the community to donate small quantities of permissively licensed text and annotations could help to address this problem. E.g. the platform could be used as part of exercises in linguistics classes at university. Students, lecturers, and researchers could contribute their own texts and annotations. Agreement evaluations could be performed in class and could be used to discuss difficult cases and to improve the students' understanding. Automatic on-demand aggregation would help to keep the administrative overhead low and the platform content usable at all times. Quality indicators such as trust towards certain contributors (e.g. known experts in the field) and agreement thresholds as used in CSniper could be used to control the quality of corpora extracted from the platform, e.g. to train models for automatic annotation tools.

Glossary

analysis component A software tool for language analysis which has been wrapped as a reusable component for a processing framework. 2–9, 15–21, 24–37, 39–53, 55–60, 62, 63, 65–71, 73–86, 92, 96, 97, 99, 101, 102, 104–107, 113–115, 118–125, 127–138, 141–143, 145, 147, 149, 150, 163, 174–177

analysis component collection A set of analysis components which are immediately interoperable without requiring any kind of data conversion. 9, 104–106, 128–130, 148

analysis tool A standalone software for language analysis, i.e. the software is not integrated with a particular processing framework. 1, 3, 4, 6, 12, 15, 17, 18, 23, 27–29, 31, 38, 39, 103, 117, 120, 121, 123, 125, 127–134, 136, 137, 142, 147–149, 151, 155, 156, 158–160, 166, 174, 176

analysis workflow A set of analysis components that are applied to primary data in a certain order to produce an analysis result. 1, 3–9, 15, 16, 18–21, 23–34, 38–40, 43–47, 51–53, 55, 56, 58–61, 65, 77–82, 84, 86, 89–92, 99, 102, 107, 113, 121, 123, 128, 130, 134, 136, 141, 142, 148, 153, 170, 171, 173–178

annotation editor An application which allows conducting a manual analysis of language data by inspecting and editing annotations over primary data, such as text. 1, 3, 6, 9, 13, 14, 18, 103–106, 116, 119, 121, 124–126, 154, 156, 157, 176, 177

annotation tool A general term for tools allowing users to create and interact with annotations, such as annotation editors, annotation visualizers, or linguistic search engines. 13, 45, 69, 103, 119, 158, 177, 178

annotation type A type to distinguish between different kinds of annotations, bearing different attributes and semantics. For example, when an annotation is realized as feature structure, the type defines which features can be used. A feature structure of the type *PartOfSpeech* may provide a feature *posTag*. 6, 17, 34, 39, 84, 103, 105, 107, 116, 120–122, 125, 134–136, 143, 147

annotation type system A set of annotation types which often interact, e.g. one type bears a feature whose value is an annotation of another type. 3, 8, 9, 15, 17, 20, 39, 101–106, 120, 128, 131

coordinate A scheme for addressing data based on key-value pairs. Maven employs GAV (*group*, *artifact*, *version*) coordinates to address artifacts. In this work, we propose a base coordinate system consisting of *type*, *language*, *variant*, and *version* for addressing resources needed by analysis components, e.g. models, dictionaries, etc. Additional coordinates like *platform* or *tag set* may be used to further qualify certain kinds of resources, such as platform-specific binaries or annotation type mappings. 35–38, 41–43, 45, 69, 140

descriptor The metadata of an item such as analysis component, analysis workflow, resource, etc. which a processing framework requires to use the item. Sometimes this is also called *description*, in particular in the context of UIMA. 19, 20, 33–35, 49, 50, 52–54, 69, 71–75, 85

feature A feature is a key/value pair used to annotate primary data. It is usually part of a feature structure, but we also use the term for attributes of annotations in general. A primitive feature has a simple value (e.g. a number or string), whereas a complex feature takes a feature structure as its value. Features are typically typed. 14, 15, 17, 101, 106, 108–111, 113–119, 122–124, 135, 136, 143, 147, 164, 165

feature structure A feature structure (FS) is a typed container for key-value pairs. The keys are strings representing the names of the features. The values are either primitive (e.g. numeric, string, etc.) or a reference to another FS. 101, 106, 109, 111, 114, 115

primary data The data being subject to analysis, e.g. text documents. In an annotated corpus, the primary data is only the text without any annotations. 4, 5, 18, 47, 59, 68, 106, 108, 112, 113, 121, 135

principle of least surprise The principle of least surprise is a general principle of design. It simply states that the user should not be surprised, e.g. by inconsistent design or unexpected behavior. The use of common terminology in scientific literature is one application of this principle. Having to press a button with the label *Start* to turn a computer off is a counterexample. (Also known as the *Law of Least Astonishment* in [123]). 137

processing framework A piece of software which enables the interoperability of analysis components and facilitates their use. The framework defines a life cycle for analysis components, means of configuring analysis components, as well as a common data structure which all analysis components use to exchange data with each other. Beyond this, a processing framework may offer many additional features, e.g. different workflow strategies, the ability to scale the processing to large data and over multiple computers, a serialization format for analyzed data, etc. 4, 5, 7, 8, 15–18, 23, 26, 28, 32, 34, 40–42, 44, 46–49, 51, 56–59, 61, 66, 68, 69, 71, 72, 75, 76, 78, 79, 81–83, 86, 90, 96, 97, 99, 101, 102, 108, 114, 121, 127–130, 132, 148, 174–176

repository A repository is a central place used to archive and share the components and resources used by an analysis workflow, their dependencies, and possibly the workflow itself. 5, 19, 29, 37, 38, 41, 43, 45, 61, 62, 65–67, 69, 71, 72, 74, 78, 147

resource Many analysis tools require resources, such as statistical or probabilistic models, dictionaries, word lists, etc. 2, 4, 5, 9, 16, 18, 20, 25–45, 47, 49, 59, 63–71, 73, 74, 76, 78, 82, 107, 128, 130, 133–135, 138–141, 147, 148, 150, 174–177

structural base type Types used to represent a structure in a generic way, without any relation to a linguistic theory or other domain, such as medicine. Domain-specific types are derived from these generic types, e.g. types for the annotation of dependency relations (e.g. *DependencyRelation*) could be derived from structural base types for relations (e.g. *Relation*). 108, 110, 111

List of Figures

List of Tables

Listings

Bibliography

[1] Anne Abeillé and Lionel Clément. Annotation morpho-syntaxique. Technical report, LLF, Université Paris 7, Jan 2003.

[2] Anne Abeillé, François Toussenel, and Martine Chéradame. Corpus le monde - annotations en constituants - guide pour les correcteurs. Technical report, LLF, UFRL, Paris 7, Mar 2004.

[3] David Abramson, Blair Bethwaite, Colin Enticott, Slavisa Garic, and Tom Peachey. Parameter space exploration using scientific workflows. In Gabrielle Allen, Jaroslaw Nabrzyski, Edward Seidel, Geert van Albada, Jack Dongarra, and Peter Sloot, editors, *Computational Science – ICCS 2009*, volume 5544 of *Lecture Notes in Computer Science*, pages 104–113. Springer Berlin / Heidelberg, 2009. URL http://dx.doi.org/10.1007/978-3-642-01970-8_11.

[4] Enrique Alfonseca, Slaven Bilac, and Stefan Pharies. German decompounding in a difficult corpus. In Alexander Gelbukh, editor, *Computational Linguistics and Intelligent Text Processing*, volume 4919 of *Lecture Notes in Computer Science*, pages 128–139. Springer Berlin / Heidelberg, 2008. ISBN 978-3-540-78134-9. URL http://dx.doi.org/10.1007/978-3-540-78135-6_12.

[5] Björkelund Anders, Bohnet Bernd, Love Hafdell, and Pierre Nugues. A high-performance syntactic and semantic dependency parser. In *Coling 2010: Demonstrations*, pages 33–36, Beijing, China, August 2010. Coling 2010 Organizing Committee. URL http://www.aclweb.org/anthology/C10-3009.

[6] Apache Ant. Java library and command-line tool whose mission is to drive processes described in build files as targets and extension points dependent upon each other. URL http://ant.apache.org (Last accessed: 2013-10-02).

[7] Apache Archiva. The Build Artifact Repository Manager. URL http://archiva.apache.org (Last accessed: 2013-11-01).

[8] Apache Hadoop. Machine learning based toolkit for the processing of natural language text. URL http://hadoop.apache.org (Last accessed: 2013-10-16).

[9] Apache OpenNLP. Machine learning based toolkit for the processing of natural language text. URL http://opennlp.apache.org (Last accessed: 2013-06-18).

[10] Apache UIMA. Apache UIMA. URL http://uima.apache.org (Last accessed: 2013-10-16).

[11] Apache UIMA-AS. Apache UIMA Asynchronous Scaleout. URL http://uima.apache.org/doc-uimaas-what.html (Last accessed: 2013-10-16).

[12] Apache UIMA Community. Apache uimaFIT guide and reference, version 2.0.0. Technical report, Apache UIMA, 2013.

[13] Apache UIMA Community. UIMA tools guide and reference, version 2.4.2. Technical report, Apache UIMA, 2013.

[14] Apache uimaFIT. Apache uimaFIT. URL http://uima.apache.org/uimafit (Last accessed: 2013-10-16).

[15] Niraj Aswani, Valentin Tablan, Kalina Bontcheva, and Hamish Cunningham. Indexing and querying linguistic metadata and document content. In *Proceedings of Fifth International Conference on Recent Advances in Natural Language Processing (RANLP2005)*, Borovets, Bulgaria, 2005.

[16] Petra Saskia Bayerl and Karsten Ingmar Paul. What determines inter-coder agreement in manual annotations? A meta-analytic investigation. *Comput. Linguist.*, 37(4):699–725, December 2011. ISSN 0891-2017. doi: 10.1162/COLI_a_00074. URL http://dx.doi.org/10.1162/COLI_a_00074.

[17] David Beaver, Itamar Francez, and Dmitry Levinson. Bad subject: (Non-) canonicality and NP distribution in existentials. In *Proceedings of SALT*, volume 15, pages 19–43, 2005.

[18] Darina Benikova, Chris Biemann, and Marc Reznicek. NoSta-D Named Entity Annotation for German: Guidelines and Dataset. In Nicoletta Calzolari, Khalid Choukri, Thierry Declerck, Hrafn Loftsson, Bente Maegaard, Joseph Mariani, Asuncion Moreno, Jan Odijk, and Stelios Piperidis, editors, *Proceedings of the Ninth International Conference on Language Resources and Evaluation (LREC'14)*, pages 2524–2531, Reykjavik, Iceland, May 2014. European Language Resources Association (ELRA). ISBN 978-2-9517408-8-4.

[19] Jon Bentley. Programming pearls: little languages. *Communications of the ACM*, 29(8): 711–721, 1986.

[20] Sandra Bergmann, Mathilde Romberg, Alexander Klenner, Christian Janßen, Thorsten Bathelt, and Guy Lonsdale. UIMA-HPC – application support and speed-up of data extraction workflows through UNICORE. In Paul Cunningham and Miriam Cunningham, editors, *eChallenges e-2012 Conference Proceedings*. IIMC International Information Management Corporation,, 2012. ISBN 978-1-905824-35-9.

[21] BerkeleyParser. A natural language parser from UC Berkeley. URL http://code.google.com/p/berkeleyparser/ (Last accessed: 2013-09-27).

[22] Steven Bethard. ClearTK-TimeML: A minimalist approach to TempEval 2013. In *Second Joint Conference on Lexical and Computational Semantics (*SEM), Volume 2: Proceedings of the Seventh International Workshop on Semantic Evaluation (SemEval 2013)*, pages 10–14, Atlanta, Georgia, USA, June 2013. Association for Computational Linguistics. URL http://www.aclweb.org/anthology/S13-2002.

[23] Ann Bies. Bies mapping for Penn Arabic treebank part-of-speech tags. URL http://www.ircs.upenn.edu/arabic/Jan03release/arabic-POStags-collapse-to-PennPOStags.txt (Last accessed: 2013-10-04), Jan 2003.

[24] Ann Bies, Mark Ferguson, Karen Katz, and Robert MacIntyre. Bracketing guidelines for Treebank II style Penn treebank project. Technical report, Linguistic Data Consortium, Jan 1995.

[25] Steven Bird and Mark Liberman. A formal framework for linguistic annotation (revised version). *Speech Communication*, 33(1-2):23–60, 2000. URL http://www.citebase.org/abstract?id=oai:arXiv.org:cs/0010033.

[26] Steven Bird, Ewan Klein, and Edward Loper. *Natural language processing with Python*. O'Reilly Media, Inc., 2009.

[27] Paul Biron, Ashok Malhotra, World Wide Web Consortium, et al. XML Schema Part 2: Datatypes Second Edition. *World Wide Web Consortium Recommendation REC-xmlschema-2-20041028*, 2004. URL http://www.w3.org/TR/2004/REC-xmlschema-2-20041028/.

[28] BNC Consortium. The British National Corpus, version 3 (BNC XML Edition). Distributed by Oxford University Computing Services p.p. the BNC Consortium, http://www.natcorp.ox.ac.uk/, 2007.

[29] K. Bontcheva, Hamish Cunningham, I. Roberts, and V. Tablan. Web-based collaborative corpus annotation: Requirements and a framework implementation. In *Proceedings of the New Challenges for NLP Frameworks Workshop at LREC*, Malta, May 2010.

[30] Kalina Bontcheva, Hamish Cunningham, Ian Roberts, Angus Roberts, Valentin Tablan, Niraj Aswani, and Genevieve Gorrell. GATE Teamware: a web-based, collaborative text annotation framework. pages 1–23, 2013. doi: 10.1007/s10579-013-9215-6. URL http://dx.doi.org/10.1007/s10579-013-9215-6.

[31] Cristina Bosco, Simonetta Montemagni, and Maria Simi. Converting Italian treebanks: Towards an Italian Stanford dependency treebank. In *Proceedings of the 7th Linguistic Annotation Workshop and Interoperability with Discourse*, pages 61–69, Sofia, Bulgaria, August 2013. Association for Computational Linguistics. URL http://www.aclweb.org/anthology/W13-2308.

[32] Thorsten Brants and Alex Franz. Web 1T 5-gram, 10 european languages version 1. Technical report, Linguistic Data Consortium, Philadelphia, 2009.

[33] Thorsten Brants, Roland Hendriks, Sabine Kramp, Brigitte Krenn, Cordula Preis, Wojciech Skut, and Hans Uszkoreit. Das NEGRA-Annotationsschema. Negra project report, Universität des Saarlandes, Saarbrücken, 1997. URL http://www.coli.uni-sb.de/sfb378/negra-corpus/negra-corpus.html.

[34] Daan Broeder, Oliver Schonefeld, Thorsten Trippel, Dieter van Uytvanck, and Andreas Witt. A pragmatic approach to XML interoperability — the component metadata infrastructure (CMDI). In *Proceedings of Balisage: The Markup Conference 2011*, volume 7 of *Balisage Series on Markup Technologies*, August 2011. doi: 10.4242/BalisageVol7.Broeder01. URL http://balisage.net/Proceedings/vol7/html/Broeder01/BalisageVol7-Broeder01.html.

[35] Sabine Buchholz and Erwin Marsi. CoNLL-X shared task on multilingual dependency parsing. In *Proceedings of the Tenth Conference on Computational Natural Language Learning*, CoNLL-X '06, pages 149–164, Stroudsburg, PA, USA, 2006. Association for Computational Linguistics. URL http://dl.acm.org/citation.cfm?id=1596276.1596305.

[36] Aljoscha Burchardt, Katrin Erk, Anette Frank, Andrea Kowalski, and Sebastian Pado. SALTO: A versatile multi-level annotation tool. In *Proceedings of the 5th international conference on language resources and evaluation (LREC 2006)*, Genoa, Italy, 2006.

[37] Aljoscha Burchardt, Katrin Erk, Anette Frank, Andrea Kowalski, Sebastian Padó, and Manfred Pinkal. The SALSA corpus: a German corpus resource for lexical semantics. In *Proceedings of the 5th International Conference on Language Resources and Evaluation (LREC-2006)*, pages 969–974, Genoa, Italy, 2006.

[38] Ekaterina Buyko and Udo Hahn. Fully embedded type systems for the semantic annotation layer. In *ICGL 2008 - Proceedings of First International Conference on Global Interoperability for Language Resources*, pages 26–33, Hong Kong, 2008.

[39] Marie Candito, Benoît Crabbé, and Mathieu Falco. Dépendances syntaxiques de surface pour le français (v1.2). Technical report, May 2011.

[40] Jean Carletta. Assessing agreement on classification tasks: the kappa statistic. *Computational linguistics*, 22(2):249–254, 1996.

[41] Jean Carletta, Jonathan Kilgour, Tim O'Donnell, Stefan Evert, and Holger Voormann. The NITE Object Model Library for handling structural linguistic annotation on multimodal data sets. In *Proceedings of the EACL Workshop on Language Technology and the Semantic Web (3rd Workshop on NLP and XML, NLPXML-2003*, Budapest, Hungary, 2003.

[42] Pi-Chuan Chang, Huihsin Tseng, Dan Jurafsky, and Christopher D. Manning. Discriminative reordering with Chinese grammatical relations features. In *Proceedings of the Third Workshop on Syntax and Structure in Statistical Translation*, SSST '09, pages 51–59, Stroudsburg, PA, USA, 2009. Association for Computational Linguistics. ISBN 978-1-932432-39-8. URL http://dl.acm.org/citation.cfm?id=1626344.1626351.

[43] Christian Chiarcos. Ontologies of linguistic annotation: Survey and perspectives. In Nicoletta Calzolari, Khalid Choukri, Thierry Declerck, Mehmet Uğur Doğan, Bente Maegaard, Joseph Mariani, Jan Odijk, and Stelios Piperidis, editors, *Proceedings of the Eight International Conference on Language Resources and Evaluation (LREC'12)*, Istanbul, Turkey, may 2012. European Language Resources Association (ELRA). ISBN 978-2-9517408-7-7.

[44] Christian Chiarcos, Stefanie Dipper, Michael Götze, Ulf Leser, Anke Lüdeling, Julia Ritz, and Manfred Stede. A flexible framework for integrating annotations from different tools and tagsets. *Traitement Automatique des Langues*, 49(2):271–293, 2008.

[45] Fernando Chirigati, Dennis Shasha, and Juliana Freire. Packing experiments for sharing and publication. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2013*, pages 977–980, New York, NY, USA, June 2013. ACM.

[46] Jinho D. Choi and Martha Palmer. Guidelines for the CLEAR style constituent to dependency conversion. Technical report 01-12, University of Colorado Boulder, Boulder, Colorado, 2012.

[47] N. Chomsky. *Aspects of the Theory of Syntax*. The MIT Press Paperback Series. Mit Press, 1965. ISBN 9780262530071. URL http://books.google.de/books?id=u0ksbFqagU8C.

[48] ClearNLP. Fast and robust NLP components implemented in Java. URL http://opennlp.apache.org (Last accessed: 2013-09-26).

[49] Benoît Crabbé and Marie Candito. Expériences d'analyse syntaxique statistique du français. In *Proceedings of TALN 2008*, Avignon, France, Jun 2008.

[50] CSniper. Combining search and annotation on large corpora. URL http://code.google.com/p/csniper/ (Last accessed: 2013-10-10).

[51] Hamish Cunningham. *Software Architecture for Language Engineering*. PhD thesis, University of Sheffield, 2000. URL http://gate.ac.uk/sale/thesis/.

[52] Hamish Cunningham, Diana Maynard, and Valentin Tablan. JAPE: a Java annotation patterns engine (second edition). Research Memorandum CS–00–10, Department of Computer Science, University of Sheffield, November 2000.

[53] Hamish Cunningham, Diana Maynard, Kalina Bontcheva, and Valentin Tablan. GATE: an architecture for development of robust HLT applications. In *Proceedings of 40th Annual Meeting of the Association for Computational Linguistics*, pages 168–175, Philadelphia, Pennsylvania, USA, July 2002. Association for Computational Linguistics. doi: 10.3115/1073083.1073112. URL http://www.aclweb.org/anthology/P02-1022.

[54] Hamish Cunningham, Valentin Tablan, Ian Roberts, Mark A. Greenwood, and Niraj Aswani. Information extraction and semantic annotation for multi-paradigm information management. In Mihai Lupu, Katja Mayer, John Tait, and Anthony J. Trippe, editors, *Current Challenges in Patent Information Retrieval*, volume 29 of *The Information Retrieval Series*. Springer, 2011.

[55] Andrew Davison. Automated capture of experiment context for easier reproducibility in computational research. *Computing in Science & Engineering*, 14(4):48–56, 2012.

[56] Johannes Daxenberger and Iryna Gurevych. Automatically classifying edit categories in Wikipedia revisions. In *Conference on Empirical Methods in Natural Language Processing (EMNLP 2013)*, pages 578–589, Stroudsburg, PA, USA, October 2013. Association for Computational Linguistics.

[57] Marie-Catherine de Marneffe and Christopher D. Manning. The Stanford typed dependencies representation. In *Coling 2008: Proceedings of the workshop on Cross-Framework and Cross-Domain Parser Evaluation*, CrossParser '08, pages 1–8, Stroudsburg, PA, USA, 2008. Association for Computational Linguistics. ISBN 978-1-905593-50-7. URL http://dl.acm.org/citation.cfm?id=1608858.1608859.

[58] J Des Rivières and J Wiegand. Eclipse: a platform for integrating development tools. *IBM Systems Journal*, 43(2):371–383, 2004.

[59] S. Dipper, M. Götze, and M. Stede. Simple annotation tools for complex annotation tasks: an evaluation. In *Proceedings of the LREC Workshop on XML-based Richly Annotated Corpora*, pages 54–62, Lisbon, Portugal, 2004.

[60] Stefanie Dipper, Michael Götze, and Stavros Skopeteas. Towards user-adaptive annotation guidelines. pages 23–30, 2004.

[61] DKPro BigData. DKPro BigData. URL http://code.google.com/p/dkpro-bigdata/ (Last accessed: 2013-07-07), 2013.

[62] DKPro Core. DKPro Core. URL http://code.google.com/p/dkpro-core-asl/ (Last accessed: 2013-12-11).

[63] DKPro Lab. DKPro Lab. URL http://code.google.com/p/dkpro-lab/ (Last accessed: 2013-12-11).

[64] DKPro Spelling. DKPro Spelling. URL http://code.google.com/p/dkpro-spelling-asl/ (Last accessed: 2013-11-11), 2013.

[65] DKPro TC. DKPro Text Classification. URL http://code.google.com/p/dkpro-tc/ (Last accessed: 2013-08-15), 2013.

[66] George Doddington, Alexis Mitchell, Mark Przybocki, Lance Ramshaw, Stephanie Strassel, and Ralph Weischedel. The automatic content extraction (ACE) program: Tasks, data, & evaluation. In *Proceedings of the 4th International Conference on Language Resources and Evaluation (LREC 2004)*, pages 837–840, Lisbon, Portugal, May 2004.

[67] Chris Drummond. Replicability is not reproducibility: nor is it good science. 2009.

[68] Chris Drummond. Reproducible research: a dissenting opinion. 2012.

[69] Richard Eckart. A framework for storing, managing and querying multi-layer annotated corpora. Diploma thesis, Technische Universität Darmstadt, Darmstadt, July 2006.

[70] Richard Eckart. Choosing an XML database for linguistically annotated corpora. In *SDV. Sprache und Datenverarbeitung 32.1/2008: International Journal for Language Data Processing*, pages 7–22, Berlin, Germany, September 2008. Workshop Datenbanktechnologien für hypermediale linguistische Anwendungen. Herbsttagung der Gesellschaft für Linguistische Datenverarbeitung (GLDV), KONVENS 2008, Berlin. ISBN Universitätsverlag Rhein-Ruhr.

[71] Richard Eckart de Castilho and Iryna Gurevych. DKPro-UGD: A flexible data-cleansing approach to processing user-generated discourse. In *Proceedings of the First French-speaking meeting around the framework Apache UIMA*, Nantes, France, July 2009. LINA CNRS UMR 6241 - University of Nantes.

[72] Richard Eckart de Castilho and Iryna Gurevych. A lightweight framework for reproducible parameter sweeping in information retrieval. In *Proceedings of the 2011 workshop on Data infrastructurEs for supporting information retrieval evaluation*, DESIRE '11, pages 7–10, New York, NY, USA, October 2011. ACM. ISBN 978-1-4503-0952-3. doi: 10.1145/2064227.2064248. URL http://doi.acm.org/10.1145/2064227.2064248.

[73] Richard Eckart de Castilho and Iryna Gurevych. Semantic service retrieval based on natural language querying and semantic similarity. In *Proceedings of the 5th IEEE International Conference on Semantic Computing (IEEE-ICSC)*, pages 173–176, Palo Alto, CA, USA, Sep 2011. doi: 10.1109/ICSC.2011.44.

[74] Richard Eckart de Castilho, Mônica Holtz, and Elke Teich. Computational support for corpus analysis work flows: The case of integrating automatic and manual annotations. In *Lingustic Processing Pipelines Workshop at GSCL 2009 - Book of Abstracts (electronic proceedings)*, September 2009.

[75] Richard Eckart de Castilho, Sabine Bartsch, and Iryna Gurevych. CSniper - annotation-by-query for non-canonical constructions in large corpora. In *Proceedings of the ACL 2012 System Demonstrations*, pages 85–90, Jeju Island, Korea, July 2012. Association for Computational Linguistics. URL http://www.aclweb.org/anthology/P12-3015.

[76] Eva Ejerhed, Gunnel Källgren, Ola Wennstedt, and Magnus Åström. The linguistic annotation system of the Stockholm-Umeå corpus project. Technical report 33, Department of Linguistics, University of Umeå, Umeå, 1992.

[77] Stefan Evert and Andrew Hardie. Twenty-first century corpus workbench: Updating a query architecture for the new millennium. In *Proceedings of the Corpus Linguistics 2011 conference*, Birmingham, UK, July 2011. University of Birmingham.

[78] EXCITEMENT. EXCITEMENT - EXploring Customer Interactions through Textual EntailMENT. URL http://www.excitement-project.eu (Last accessed: 2013-12-11).

[79] Fairview Research LLC. User guide for version 1.3 of GATE Teamware (draft). URL http://gate.ac.uk/teamware/teamware-1.3-guide.pdf (Last accessed: 2013-10-16), April 2010.

[80] Xubo Fei and Shiyong Lu. A dataflow-based scientific workflow composition framework. *Services Computing, IEEE Transactions on*, 5(1):45–58, 2012.

[81] D. Ferrucci and A. Lally. Building an example application with the unstructured information management architecture. *IBM Syst. J.*, 43:455–475, July 2004. ISSN 0018-8670. doi: http://dx.doi.org/10.1147/sj.433.0455. URL http://dx.doi.org/10.1147/sj.433.0455.

[82] D.A. Ferrucci. Introduction to "This is Watson". *IBM Journal of Research and Development*, 56(3.4):1:1–1:15, 2012. ISSN 0018-8646. doi: 10.1147/JRD.2012.2184356.

[83] David Ferrucci and Adam Lally. UIMA: an architectural approach to unstructured information processing in the corporate research environment. *Natural Language Engineering*, 10(3-4):327–348, 2004. doi: 10.1017/S1351324904003523. URL http://dx.doi.org/10.1017/S1351324904003523.

[84] David Ferrucci, Adam Lally, Karin Verspoor, and Eric Nyberg. Unstructured information management architecture (UIMA) version 1.0. OASIS Standard, March 2009.

[85] Oliver Ferschke, Iryna Gurevych, and Marc Rittberger. FlawFinder: A modular system for predicting quality flaws in Wikipedia - Notebook for PAN at CLEF 2012. In Pamela Forner, Jussi Karlgren, and Christa Womser-Hacker, editors, *CLEF 2012 Labs and Workshop, Notebook Papers*, September 2012. ISBN 978-88-904810-3-1.

[86] Jenny Rose Finkel, Trond Grenager, and Christopher Manning. Incorporating non-local information into information extraction systems by gibbs sampling. In *Proceedings of the 43rd Annual Meeting on Association for Computational Linguistics*, ACL '05, pages 363–370, Stroudsburg, PA, USA, 2005. Association for Computational Linguistics. doi: 10.3115/1219840.1219885. URL http://dx.doi.org/10.3115/1219840.1219885.

[87] Lucie Flekova and Iryna Gurevych. Can we hide in the web? large scale simultaneous age and gender author profiling in social media - Notebook for PAN at CLEF 2013 scale simultaneous age and gender author profiling in social media - notebook for PAN at CLEF 2013. In Dan Tufis Pamela Forner, Roberto Navigli, editor, *CLEF 2013 Labs and Workshops - Notebook Papers*, September 2013. ISBN 978-88-904810-5-5.

[88] Karën Fort, Adeline Nazarenko, and Sophie Rosset. Modeling the complexity of manual annotation tasks: a grid of analysis. In *Proceedings of the International Conference on Computational Linguistics (COLING 2012)*, pages 895–910, Mumbaï, India, December 2012. URL http://hal.archives-ouvertes.fr/hal-00769631. Quaero.

[89] Martin Fowler. FluentInterface. URL http://martinfowler.com/bliki/FluentInterface.html (Last accessed: 2013-06-18), December 2005.

[90] Martin Fowler. DslBoundary. URL http://martinfowler.com/bliki/DslBoundary.html (Last accessed: 2013-06-18), August 2006.

[91] FreeLing. An open source suite of language analyzers. URL http://nlp.lsi.upc.edu/freeling/ (Last accessed: 2013-09-29).

[92] Juliana Freire and Claudio T. Silva. Making computations and publications reproducible with VisTrails. *Computing in Science Engineering*, 14(4):18–25, 2012. ISSN 1521-9615. doi: 10.1109/MCSE.2012.76.

[93] Cláudia Freitas and Susana Afonso. Bíblia florestal: Um manual lingüístico da floresta sintá(c)tica. Technical report, http://www.linguateca.pt, Sep 2008.

[94] Lauren Friedman, Lee Haejoong, and Stephanie Strassel. Control Framework for Gold Standard Reference Translations: The Process and Toolkit Developed for GALE. In *Translating and the Computer 30*, London, UK, November 2008.

[95] Pablo Gamallo. Tag set for Portuguese used by TreeTagger. URL http://gramatica.usc.es/~gamallo/tagger.htm (Last accessed: 2013-10-04), Dec 2005.

[96] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design patterns: Abstraction and reuse of object-oriented design*. Springer, 2001.

[97] Elmer Garduno, Zi Yang, Avner Maiberg, Collin McCormack, Yan Fang, and Eric Nyberg. CSE framework: A UIMA-based distributed system for configuration space exploration. In Peter Kluegl, Richard Eckart de Castilho, and Katrin Tomanek, editors, *Proceedings of the 3rd UIMAGSCL Workshop*, pages 14–17, Darmstadt, Germany, September 2013.

[98] GATE. General Architecture for Text Engineering. URL http://gate.ac.uk (Last accessed: 2013-09-26).

[99] Joachim Gauck, Angela Merkel, and Sabine Leutheusser-Schnarrenberger. Achtes Gesetz zur Änderung des Urheberrechtsgesetzes. *Bundesgesetzblatt*, I(23):1161, May 2013.

[100] GENIATagger. Part-of-speech tagging, shallow parsing, and named entity recognition for biomedical text. URL http://www.nactem.ac.uk/GENIA/tagger/ (Last accessed: 2013-09-29).

[101] Sumukh Ghodke and Steven Bird. Fangorn: A system for querying very large treebanks. In *Proceedings of COLING 2012: Demonstration Papers*, pages 175–182, Mumbai, India, December 2012. The COLING 2012 Organizing Committee. URL http://www.aclweb.org/anthology/C12-3022.

[102] Alfio Gliozzo, Chris Biemann, Martin Riedl, Bonaventura Coppola, Michael R. Glass, and Matthew Hatem. Jobimtext visualizer: A graph-based approach to contextualizing distributional similarity. In *Proceedings of the 8th Workshop on TextGraphs in conjunction with EMNLP 2013*, Seattle, WA, USA, 2013.

[103] Carole A Goble, Jiten Bhagat, Sergejs Aleksejevs, Don Cruickshank, Danius Michaelides, David Newman, Mark Borkum, Sean Bechhofer, Marco Roos, Peter Li, et al. myExperiment: a repository and social network for the sharing of bioinformatics workflows. *Nucleic acids research*, 38(suppl 2):W677–W682, 2010.

[104] José Manuel Gómez-Pérez, Esteban Garcıa-Cuesta, Jun Zhao, Aleix Garrido, José Enrique Ruiz, and Graham Klyne. How reliable is your workflow: Monitoring decay in scholarly publications. In *Proceedings of the 3rd Workshop on Semantic Publishing (SePublica 2013) at 10th Extended Semantic Web Conference*, page 75, Montpellier, France, May 2013.

[105] T. Götz and O. Suhre. Design and implementation of the UIMA common analysis system. *IBM Systems Journal*, 43(3):476 –489, 2004. ISSN 0018-8670. doi: 10.1147/sj.433.0476.

[106] Philip J. Guo and Dawson Engler. CDE: using system call interposition to automatically create portable software packages. In *Proceedings of the 2011 USENIX conference on USENIX annual technical conference*, USENIXATC'11, pages 21–21, Berkeley, CA, USA, 2011. USENIX Association. URL http://dl.acm.org/citation.cfm?id=2002181.2002202.

[107] Iryna Gurevych, Max Mühlhäuser, Christof Müller, Jürgen Steimle, Markus Weimer, and Torsten Zesch. Darmstadt Knowledge Processing Repository based on UIMA. In *Proceedings of the First Workshop on Unstructured Information Management Architecture at Biannual Conference of the Society for Computational Linguistics and Language Technology*, Tübingen, Germany, April 2007.

[108] Udo Hahn, Ekaterina Buyko, Katrin Tomanek, Scott Piao, John McNaught, Yoshimasa Tsuruoka, and Sophia Ananiadou. An annotation type system for a data-driven NLP pipeline. In *Proceedings of the Linguistic Annotation Workshop*, LAW '07, pages 33–40, Stroudsburg, PA, USA, 2007. Association for Computational Linguistics. URL http://dl.acm.org/citation.cfm?id=1642059.1642064.

[109] Udo Hahn, Ekaterina Buyko, Rico Landefelda, Matthias Mühlhausen, Michael Poprat, Katrin Tomanek, and Joachim Wermter. An overview of JCoRe, the JULIE Lab UIMA component repository. In *Proceedings of the LREC'08 Workshop "Towards Enhanced Interoperability for Large HLT Systems: UIMA for NLP"*, pages 1–7, Marrakech, Morocco, May 2008.

[110] Mark Hall, Eibe Frank, Geoffrey Holmes, Bernhard Pfahringer, Peter Reutemann, and Ian H. Witten. The WEKA data mining software: an update. *SIGKDD Explor. Newsl.*, 11 (1):10–18, November 2009. ISSN 1931-0145. doi: 10.1145/1656274.1656278. URL http://doi.acm.org/10.1145/1656274.1656278.

[111] Michael A. K. Halliday and Christian M. I. M. Matthiessen. *An introduction to functional grammar, 3rd Edition*. Arnold Publishers, London, UK, 2004.

[112] Jürgen Hermes. *Textprozessierung - Design und Applikation*. PhD thesis, Universität zu Köln, February 2012. URL http://kups.ub.uni-koeln.de/4561/.

[113] Nicolas Hernandez. Tackling interoperability issues within UIMA workflows. In Nicoletta Calzolari, Khalid Choukri, Thierry Declerck, Mehmet Uğur Doğan, Bente Maegaard, Joseph Mariani, Jan Odijk, and Stelios Piperidis, editors, *Proceedings of the Eight International Conference on Language Resources and Evaluation (LREC'12)*, Istanbul, Turkey, may 2012. European Language Resources Association (ELRA). ISBN 978-2-9517408-7-7.

[114] Kristina M Hettne, Katherine Wolstencroft, Khalid Belhajjame, Carole A Goble, Eleni Mina, Harish Dharuri, David De Roure, Lourdes Verdes-Montenegro, Julián Garrido, and Marco Roos. Best practices for workflow design: how to prevent workflow decay. In Paschke Adrian, Albert Burger, Paolo Romano, M. Scott Marshall, and Andrea Splendiani, editors, *Proceedings of the 5th International Workshop on Semantic Web Applications and Tools for Life Sciences (SWAT4LS 2012)*, Paris, France, November 2012.

[115] Erhard Hinrichs, Heike Neuroth, and Peter Wittenburg, editors. *Service-oriented Architectures (SOAs) for the Humanities: Solutions and Impacts – Interaction in Joint CLARIN-D/DARIAH Workshop at Digital Humanities Conference 2012*, July 2012.

[116] Marie Hinrichs, Thomas Zastrow, and Erhard Hinrichs. WebLicht: Web-based LRT Services in a Distributed eScience Infrastructure. In Nicoletta Calzolari, Khalid Choukri,

Bente Maegaard, Joseph Mariani, Jan Odijk, Stelios Piperidis, Mike Rosner, and Daniel Tapias, editors, *Proceedings of the Seventh International Conference on Language Resources and Evaluation (LREC'10)*, pages 489–493, Valletta, Malta, May 2010. European Language Resources Association (ELRA). ISBN 2-9517408-6-7.

[117] Nancy Ide and Laurent Romary. Representing linguistic corpora and their annotations. In *Proceedings of the 5th international conference on language resources and evaluation (LREC 2006)*, pages 225–228, Genoa, Italy, 2006.

[118] Nancy Ide and Keith Suderman. GrAF: A graph-based format for linguistic annotations. In *Proceedings of the Linguistic Annotation Workshop*, pages 1–8, Prague, Czech Republic, June 2007. Association for Computational Linguistics. URL http://www.aclweb.org/anthology/W/W07/W07-1501.

[119] ISO/IEC JTC 1/SC 2/WG 3. 8-bit single-byte coded graphic character sets, part 1: Latin alphabet no. 1. URL http://www.open-std.org/JTC1/SC2/WG3/docs/n411.pdf (Last accessed: 2013-06-18), April 1998.

[120] ISO/IEC19757-3. Information technology - document schema definition languages (DSDL) - part 3: Rule-based validation - Schematron. URL http://dsdl.org, June 2006.

[121] ISO/TC 37/SC 2. Codes for the representation of names of languages – part 2: Alpha-3 code. URL http://www.iso.org/iso/catalogue_detail?csnumber=4767 (Last accessed: 2013-06-19), 1988.

[122] ISO/TC 37/SC 2. Codes for the representation of names of languages – part 1: Alpha-2 code. URL http://www.iso.org/iso/catalogue_detail?csnumber=22109 (Last accessed: 2013-06-19), 2002.

[123] Geoffrey James. *The Tao of Programming*. InfoBooks, Santa Monica, CA, USA, 1987. ISBN 0931137071.

[124] Thorsten Joachims. Advances in kernel methods. chapter Making large-scale support vector machine learning practical, pages 169–184. MIT Press, Cambridge, MA, USA, 1999. ISBN 0-262-19416-3. URL http://dl.acm.org/citation.cfm?id=299094.299104.

[125] JoBimText. JoBimText. URL http://sourceforge.net/projects/jobimtext/ (Last accessed: 2013-12-11).

[126] Diane Jordan, John Evdemon, Alexandre Alves, Assaf Arkin, Sid Askary, Charlton Barreto, Ben Bloch, Francisco Curbera, Mark Ford, Yaron Goland, Alejandro Guízar, Neelakantan Kartha, Canyang Kevin Liu, Rania Khalaf, Dieter König, Mike Marin, Vinkesh Mehta, Satish Thatte, Danny van der Rijn, Prasad Yendluri, and Alex Yiu. Web services business process execution language version 2.0. OASIS Standard, December 2007. URL http://docs.oasis-open.org/wsbpel/2.0/wsbpel-v2.0.html.

[127] Matthias L. Jugel and Stephan J. Schmidt. The Radeox wiki render engine. In *Proceedings of the 2006 international symposium on Wikis*, WikiSym '06, pages 33–36, New York, NY, USA, 2006. ACM. ISBN 1-59593-413-8. doi: 10.1145/1149453.1149465. URL http://doi.acm.org/10.1145/1149453.1149465.

[128] Yoshinobu Kano, Luke McCrohon, Sophia Ananiadou, and Jun'ichi Tsujii. Integrated NLP evaluation system for pluggable evaluation metrics with extensive interoperable toolkit. In *Proceedings of the Workshop on Software Engineering, Testing, and Quality*

Assurance for Natural Language Processing, SETQA-NLP '09, pages 22–30, Stroudsburg, PA, USA, 2009. Association for Computational Linguistics. ISBN 978-1-932432-32-9. URL http://dl.acm.org/citation.cfm?id=1621947.1621951.

[129] Yoshinobu Kano, Makoto Miwa, Kevin Bretonnel Cohen, Lawrence E. Hunter, Sophia Ananiadou, and Jun'ichi Tsujii. U-Compare: A modular NLP workflow construction and evaluation system. *IBM Journal of Research and Development*, 55(3):11:1–11:10, May 2011. ISSN 0018-8646. URL http://dl.acm.org/citation.cfm?id=2001058.2001068.

[130] Marc Kemps-Snijders, Menzo Windhouwer, Peter Wittenburg, and Sue Ellen Wright. ISOcat: remodelling metadata for language resources. *Int. J. Metadata, Semantics and Ontologies*, 4(4):261–276, 2009.

[131] Dan Klein and Christopher D Manning. Fast exact inference with a factored model for natural language parsing. *Advances in Neural Information Processing Systems*, 15(2003): 3–10, 2003.

[132] Dan Klein and Christopher D. Manning. Accurate unlexicalized parsing. In *Proceedings of the 41st Annual Meeting on Association for Computational Linguistics - Volume 1*, ACL '03, pages 423–430, Stroudsburg, PA, USA, 2003. Association for Computational Linguistics. doi: 10.3115/1075096.1075150. URL http://dx.doi.org/10.3115/1075096.1075150.

[133] Peter Kluegl, Martin Atzmueller, and Frank Puppe. TextMarker: A tool for rule-based information extraction. In Christian Chiarcos, Richard Eckart de Castilho, and Manfred Stede, editors, *Proceedings of the Biennial GSCL Conference 2009, 2nd UIMA@GSCL Workshop*, pages 233–240, Potsdam, Germany, September 2009. Gunter Narr Verlag.

[134] Matthias Trautner Kromann, Line Mikkelsen, and Stine Kern Lynge. Danish dependency treebank - annotation guide. URL http://www.buch-kromann.dk/matthias/treebank/guide.html (Last accessed: 2013-09-30), Nov 2004.

[135] LanguageTool. LanguageTool style and grammar checker. URL http://www.languagetool.org (Last accessed: 2013-06-18).

[136] Martha Larson, Daniel Willett, Joachim Köhler, and Gerhard Rigoll. Compound splitting and lexical unit recombination for improved performance of a speech recognition system for German parliamentary speeches. In *Proceedings ICSLP 2000: Sixth International Conference on Spoken Language Processing*, pages 945–948, 2000.

[137] Heeyoung Lee, Angel Chang, Yves Peirsman, Nathanael Chambers, Mihai Surdeanu, and Dan Jurafsky. Deterministic coreference resolution based on entity-centric, precision-ranked rules. *Computational Linguistics*, 39(4), 2013.

[138] Wolfgang Lezius. *Ein Suchwerkzeug für syntaktisch annotierte Textkorpora (German)*. Ph.d. thesis, University of Stuttgart, Institut für Maschinelle Sprachverarbeitung, 2002. URL http://www.ims.uni-stuttgart.de/projekte/corplex/paper/lezius/diss/disslezius.pdf.

[139] Tim Lindholm, Frank Yellin, Gilad Bracha, and Alex Buckley. *The Java virtual machine specification*. Addison-Wesley, 2013.

[140] LingPipe. A tool kit for processing text using computational linguistics. URL http://alias-i.com/lingpipe/ (Last accessed: 2013-09-29).

[141] W. Lloyd, O. David, J. C. Ascough, II, K. W. Rojas, J. R. Carlson, G. H. Leavesley, P. Krause, T. R. Green, and L. R. Ahuja. Environmental modeling framework invasiveness: Analysis and implications. *Environ. Model. Softw.*, 26(10):1240–1250, October 2011. ISSN 1364-8152. doi: 10.1016/j.envsoft.2011.03.011. URL http://dx.doi.org/10.1016/j.envsoft.2011.03.011.

[142] Mohamed Maamouri, Ann Bies, Sondos Krouna, Fatma Gaddeche, and Basma Bouziri. Penn Arabic treebank guidelines (v 4.8). Technical report, Linguistic Data Consortium, University of Pennsylvania, Philadelphia, PA, Jan 2009.

[143] Mohamed Maamouri, Ann Bies, Sondos Krouna, Dalila Tabessi, Fatma Gaddeche, and Basma Bouziri. Penn Arabic treebank (PATB) guidelines - morphological and syntactic guidelines. URL http://projects.ldc.upenn.edu/ArabicTreebank/ (Last accessed: 2013-09-30), Jun 2011.

[144] MaltParser. A system for data-driven dependency parsing. URL http://www.maltparser.org (Last accessed: 2013-09-27).

[145] MAMBAlex. Lexical categories in MAMBA. URL http://stp.lingfil.uu.se/~nivre/swedish_treebank/MAMBAlex.html (Last accessed: 2013-10-04).

[146] Mitchell P. Marcus, Mary Ann Marcinkiewicz, and Beatrice Santorini. Building a large annotated corpus of English: the Penn treebank. *Comput. Linguist.*, 19(2):313–330, June 1993. ISSN 0891-2017. URL http://dl.acm.org/citation.cfm?id=972470.972475.

[147] Montserrat Marimon, Beatríz Fisas, Núria Bel, Marta Villegas, Jorge Vivaldi, Sergi Torner, Mercè Lorente, Silvia Vázquez, and Marta Villegas. The IULA treebank. In Nicoletta Calzolari, Khalid Choukri, Thierry Declerck, Mehmet Uğur Doğan, Bente Maegaard, Joseph Mariani, Jan Odijk, and Stelios Piperidis, editors, *Proceedings of the Eight International Conference on Language Resources and Evaluation (LREC'12)*, Istanbul, Turkey, may 2012. European Language Resources Association (ELRA). ISBN 978-2-9517408-7-7.

[148] Mate-Tools. Tools for natural language analysis, generation and machine learning. URL http://code.google.com/p/mate-tools/ (Last accessed: 2013-09-26).

[149] Maven Central. The Central Repository. URL http://search.maven.org (Last accessed: 2013-06-19), 2011. Sonatype Inc. (http://www.sonatype.org/central).

[150] Andrew Kachites McCallum. MALLET: A machine learning for language toolkit. URL http://mallet.cs.umass.edu (Last accessed: 2013-10-02).

[151] Ryan McDonald, Joakim Nivre, Yvonne Quirmbach-Brundage, Yoav Goldberg, Dipanjan Das, Kuzman Ganchev, Keith Hall, Slav Petrov, Hao Zhang, Oscar Täckström, Claudia Bedini, Núria Bertomeu Castelló, and Jungmee Lee. Universal dependency annotation for multilingual parsing. In *Proceedings of the 51st Annual Meeting of the Association for Computational Linguistics (Volume 2: Short Papers)*, pages 92–97, Sofia, Bulgaria, August 2013. Association for Computational Linguistics. URL http://www.aclweb.org/anthology/P13-2017.

[152] Tony McEnery and Richard Xiao. The Lancaster corpus of Mandarin Chinese (LCMC). URL http://www.lancaster.ac.uk/fass/projects/corpus/LCMC/ (Last accessed: 2013-09-30), Feb 2004.

[153] Deborah L McGuinness, Frank Van Harmelen, et al. OWL Web Ontology Language overview. *W3C recommendation*, 10(2004-03):10, 2004.

[154] David McKelvie, Amy Isard, Andreas Mengel, Morten Baun Møller, Michael Grosse, and Marion Klein. The MATE workbench - an annotation tool for XML coded speech corpora. *Speech Communication*, 33(1-2):97–112, 2001.

[155] MeCab. Japanese morphological analyzer. URL http://mecab.googlecode.com (Last accessed: 2013-09-27).

[156] Frederic P. Miller, Agnes F. Vandome, and John McBrewster. *Apache Maven*. Alphascript Publishing, 2010.

[157] George A. Miller. WordNet: a lexical database for English. *Commun. ACM*, 38(11): 39–41, November 1995. ISSN 0001-0782. doi: 10.1145/219717.219748. URL http://doi.acm.org/10.1145/219717.219748.

[158] James D. Mooney. Bringing portability to the software process. *Dept. of Statistics and Comp. Sci., West Virginia Univ., Morgantown WV*, 1997.

[159] Luc Moreau, Ben Clifford, Juliana Freire, Joe Futrelle, Yolanda Gil, Paul Groth, Natalia Kwasnikowska, Simon Miles, Paolo Missier, Jim Myers, et al. The open provenance model core specification (v1. 1). *Future Generation Computer Systems*, 27(6):743–756, 2011.

[160] R. L. Morgan, Scott Cantor, Steven Carmody, Walter Hoehn, and Ken Klingenstein. Federated security: The shibboleth approach. *Educause Quarterly*, 27(4):12–17, 2004.

[161] Morpha. Morpha lex stemmer converted using jflex. URL http://github.com/knowitall/morpha (Last accessed: 2013-09-27).

[162] Alessandro Moschitti. Making tree kernels practical for natural language learning. In *Proceedings of the Eleventh International Conference on European Association for Computational Linguistics (EACL'06)*, pages 113–120, Trento, Italy, 2006.

[163] MSTParser. A non-projective dependency parser that searches for maximum spanning trees over directed graphs. URL http://sourceforge.net/projects/mstparser/ (Last accessed: 2013-09-27).

[164] Christoph Müller and Michael Strube. Multi-level annotation of linguistic data with MMAX2. *English Corpus Linguistics, Vol.3*, pages 197–214, 2006.

[165] B Clifford Neuman and Theodore Ts'o. Kerberos: An authentication service for computer networks. *Communications Magazine, IEEE*, 32(9):33–38, 1994.

[166] Tae-Gil Noh and Sebastian Padó. Using UIMA to structure an open platform for textual entailment. In Peter Klügl, Richard Eckart de Castilho, and Katrin Tomanek, editors, *Proceedings of the 3rd Workshop on Unstructured Information Management Architecture (UIMA@GSCL 2013)*, pages 26–33, Darmstadt, Germany, Sep 2013. CEUR-WS.org.

[167] Tae-Gil Noh, Sebastian Padó, Asher Stern, Ofer Bronstein, Rui Wang, and Roberto Zanoli. EXCITEMENT open platform: Architecture and interfaces (v1.1.4). Specification, EXCITEMENT, October 2013.

[168] Mick O'Donnell. The UAM CorpusTool: Software for corpus annotation and exploration. In *Proceedings of the XXVI Congreso de AESLA*, 2008.

[169] Eduardo Ogasawara, Daniel de Oliveira, Fernando Chirigati, Carlos Eduardo Barbosa, Renato Elias, Vanessa Braganholo, Alvaro Coutinho, and Marta Mattoso. Exploring many task computing in scientific workflows. In *Proceedings of the 2nd Workshop on Many-Task*

Computing on Grids and Supercomputers, MTAGS '09, pages 2:1–2:10, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-714-1. doi: 10.1145/1646468.1646470. URL http://doi.acm.org/10.1145/1646468.1646470.

[170] Philip V. Ogren. Knowtator: a Protégé plug-in for annotated corpus construction. In *Proceedings of the 2006 Conference of the North American Chapter of the Association for Computational Linguistics on Human Language Technology: companion volume: demonstrations*, NAACL-Demonstrations '06, pages 273–275, Stroudsburg, PA, USA, 2006. Association for Computational Linguistics. doi: 10.3115/1225785.1225791. URL http://dx.doi.org/10.3115/1225785.1225791.

[171] Philip V. Ogren and Steven J. Bethard. Building test suites for UIMA components. In *Proceedings of the Workshop on Software Engineering, Testing, and Quality Assurance for Natural Language Processing (SETQA-NLP 2009)*, pages 1–4, Boulder, Colorado, June 2009. Association for Computational Linguistics. URL http://www.aclweb.org/anthology/W/W09/W09-1501.

[172] Philip V. Ogren, Philipp G. Wetzler, and Steven J. Bethard. ClearTK: a framework for statistical natural language processing. In Christian Chiarcos, Richard Eckart de Castilho, and Manfred Stede, editors, *Proceedings of the Biennial GSCL Conference 2009, 2nd UIMA@GSCL Workshop*, pages 241–248, Potsdam, Germany, September 2009. Gunter Narr Verlag.

[173] OMG. OMG XML metadata interchange (XMI) specification. Technical report, Object Management Group, Inc., January 2002.

[174] Oracle and/or its affiliates. Java object serialization specification. URL http://docs.oracle.com/javase/7/docs/platform/serialization/spec/serialTOC.html (Last accessed: 2013-06-18), 2010.

[175] Petya Osenova and Kiril Simov. Formal grammar of bulgarian language. Technical report, Institute for Parallel Processing, Bulgarian Academy of Sciences, Sofia, Bulgaria, Dec 2007.

[176] PAROLE-ES. PAROLE tagset for Spanish v2.0. URL http://nlp.lsi.upc.edu/freeling/doc/tagsets/tagset-es.html (Last accessed: 2013-10-04).

[177] PAROLE-Reduced. SVMTool - PAROLE reduced tagset. URL http://www.lsi.upc.edu/~nlp/SVMTool/parole.html (Last accessed: 2013-10-04).

[178] Slav Petrov, Dipanjan Das, and Ryan McDonald. A universal part-of-speech tagset. In Nicoletta Calzolari, Khalid Choukri, Thierry Declerck, Mehmet Uğur Doğan, Bente Maegaard, Joseph Mariani, Jan Odijk, and Stelios Piperidis, editors, *Proceedings of the Eight International Conference on Language Resources and Evaluation (LREC'12)*, Istanbul, Turkey, may 2012. European Language Resources Association (ELRA). ISBN 978-2-9517408-7-7.

[179] Scott Piao, Sophia Ananiadou, and John McNaught. Integrating annotation tools into UIMA for interoperability. In *Proceedings of the UK e-Science AHM Conference 2007*, pages 575–582, 2007.

[180] Brett Porter. Maven turns 3.0. *JAX Magazine*, 3:3–7, 2010.

[181] Martin F Porter. Snowball: A language for stemming algorithms. URL http://snowball.tartarus.org/texts/introduction.html (Last accessed: 2013-10-16), October 2001.

[182] Janina Radó. Object fronting in English and German: A quantitative corpus study. In Berry Claus, Constantin Freitag, Sophie Repp, and Edith Scheifele, editors, *Proceedings of the Linguistic Evidence Special Edition 2013*, Berlin, April 2013. URL http://www2.hu-berlin.de/linguistic-evidence-berlin-2013/download/abstracts/Rado_LinguisticEvidence2013.pdf.

[183] Rafal Rak, Andrew Rowley, William Black, and Sophia Ananiadou. Argo: an integrative, interactive, text mining-based workbench supporting curation. *Database*, 2012, 2012. doi: 10.1093/database/bas010. URL http://database.oxfordjournals.org/content/2012/bas010.abstract.

[184] Eric S Raymond. *The art of Unix programming*. Addison-Wesley Professional, 2003.

[185] Martin Riedl and Chris Biemann. Text segmentation with topic models. *JLCL*, 27(1): 47–69, 2012.

[186] Christophe Roeder, Philip V. Ogren, William A. Baumgartner Jr., and Lawrence Hunter. Simplifying UIMA component development and testing with Java annotations and dependency injection. In Christian Chiarcos, Richard Eckart de Castilho, and Manfred Stede, editors, *Proceedings of the Biennial GSCL Conference 2009, 2nd UIMA@GSCL Workshop*, pages 257–260. Gunter Narr Verlag, 2009.

[187] Douglas LT Rohde. Tgrep2 user manual version 1.15. *Massachusetts Institute of Technology. http://tedlab.mit.edu/dr/Tgrep2*, 2005.

[188] Fernando Sánchez León. A Spanish tagset for the CRATER project. Technical report, Laboratorio de Lingüística Informática, Facultad de Filosofía y Letras, Universidad Autónoma de Madrid, Jun 1994.

[189] Beatrice Santorini. Part-of-speech tagging guidelines for the Penn treebank project (3rd revision). Technical report MS-CIS-90-47, LINC LAB 178, University of Pennsylvania, Jul 1990.

[190] Guergana K. Savova, James J. Masanz, Philip V. Ogren, Jiaping Zheng, Sunghwan Sohn, Karin C. Kipper-Schuler, and Christopher G. Chute. Mayo clinical text analysis and knowledge extraction system (cTAKES): architecture, component evaluation and applications. *Journal of the American Medical Informatics Association*, 17(5):507–513, 2010.

[191] Anne Schiller, Simone Teufel, and Christine Stöckert. Guidelines für das Tagging deutscher Textcorpora mit STTS. Technical report, Universität Stuttgart, Universität Tübingen, Aug 1999.

[192] Helmut Schmid. Probabilistic part-of-speech tagging using decision trees. In *Proceedings of International Conference on New Methods in Language Processing*, pages 44–49, Manchester, UK, 1994.

[193] Marshall Schor. An effective, Java-friendly interface for the unstructured management architecture (UIMA) common analysis system. Technical Report IBM RC23176, IBM T. J. Watson Research Center, 2004.

[194] Stephan Schwiebert. *Tesla - ein virtuelles Labor für experimentelle Computer- und Korpuslinguistik*. PhD thesis, Universität zu Köln, 2012. URL http://kups.ub.uni-koeln.de/4571/.

[195] Aleksander Slominski. Adapting BPEL to scientific workflows. In Ian J. Taylor, Ewa Deelman, Dennis B. Gannon, and Matthew Shields, editors, *Workflows for e-Science*, pages 208–226. Springer London, 2007. ISBN 978-1-84628-519-6. doi: 10.1007/978-1-84628-757-2_14. URL http://dx.doi.org/10.1007/978-1-84628-757-2_14.

[196] Snowball Stemmer (Lucene). Machine learning based toolkit for the processing of natural language text. URL http://lucene.apache.org/core/3_0_3/api/contrib-snowball/ (Last accessed: 2013-09-27).

[197] Richard Socher, John Bauer, Christopher D. Manning, and Ng Andrew Y. Parsing with compositional vector grammars. In *Proceedings of the 51st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 455–465, Sofia, Bulgaria, August 2013. Association for Computational Linguistics. URL http://www.aclweb.org/anthology/P13-1045.

[198] Richard Socher, Alex Perelygin, Jean Wu, Jason Chuang, Christopher D. Manning, Andrew Y. Ng, and Christopher Potts. Recursive deep models for semantic compositionality over a sentiment treebank. In *Proceedings of the 2013 Conference on Empirical Methods in Natural Language Processing*, pages 1631–1642, Stroudsburg, PA, October 2013. Association for Computational Linguistics.

[199] Bàrbara Soriano, Oriol Borrega, Mariona Taulé, and M. Antònia Martí. Guidelines. Technical report 3LB-WP 03-02, Universitat de Barcelona, 2008.

[200] Stanford CoreNLP. A suite of core NLP tools. URL http://nlp.stanford.edu/software/corenlp.shtml (Last accessed: 2013-09-06).

[201] STB-DEP. Dependency Labels in the Swedish Treebank. URL http://stp.lingfil.uu.se/~nivre/swedish_treebank/dep.html (Last accessed: 2013-10-04).

[202] STB-POS. Part-of-speech categories in the Swedish Treebank. URL http://stp.lingfil.uu.se/~nivre/swedish_treebank/pos.html (Last accessed: 2013-10-04).

[203] Achim Stein. French TreeTagger part-of-speech tags. URL http://nlp.lsi.upc.edu/freeling/doc/tagsets/tagset-es.html (Last accessed: 2013-10-04), 2003.

[204] Pontus Stenetorp, Sampo Pyysalo, Goran Topić, Tomoko Ohta, Sophia Ananiadou, and Jun'ichi Tsujii. brat: a web-based tool for NLP-assisted text annotation. In *Proceedings of the Demonstrations at the 13th Conference of the European Chapter of the Association for Computational Linguistics*, pages 102–107, Avignon, France, April 2012. Association for Computational Linguistics. URL http://www.aclweb.org/anthology/E12-2021.

[205] Victoria Stodden. The legal framework for reproducible scientific research: Licensing and copyright. *Computing in Science & Engineering*, 11(1):35–40, 2009.

[206] Victoria Stodden. Reproducible research: Addressing the need for data and code sharing in computational science. *Computing in Science & Engineering*, 12(5):8–12, 2010.

[207] Jannik Strötgen and Michael Gertz. Multilingual and cross-domain temporal tagging. *Language Resources and Evaluation*, 47(2):269–298, 2013. doi: 10.1007/s10579-012-9179-y.

[208] Mihai Surdeanu, Richard Johansson, Adam Meyers, Lluís Màrquez, and Joakim Nivre. The CoNLL 2008 shared task on joint parsing of syntactic and semantic dependencies. In *CoNLL 2008: Proceedings of the Twelfth Conference on Computational Natural Language*

Learning, pages 159–177, Manchester, England, August 2008. Coling 2008 Organizing Committee. URL http://www.aclweb.org/anthology/W08-2121.

[209] Valentin Tablan, Ian Roberts, Hamish Cunningham, and Kalina Bontcheva. GATE-Cloud.net: a platform for large-scale, open-source text processing on the cloud. *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences*, 371(1983), 2013.

[210] Antero Taivalsaari. On the notion of inheritance. *ACM Computing Surveys (CSUR)*, 28 (3):438–479, 1996.

[211] TANL-IT. Tanl part-of-speech tagset. URL http://medialab.di.unipi.it/wiki/Tanl_POS_Tagset (Last accessed: 2013-10-04).

[212] Mariona Taulé, M. Antònia Martí, and Marta Recasens. AnCora: Multilevel annotated corpora for Catalan and Spanish. In Nicoletta Calzolari, Khalid Choukri, Bente Maegaard, Joseph Mariani, Jan Odijk, Stelios Piperidis, and Daniel Tapias, editors, *Proceedings of the Sixth International Conference on Language Resources and Evaluation (LREC'08)*, Marrakech, Morocco, May 2008. European Language Resources Association (ELRA). ISBN 2-9517408-4-0. http://www.lrec-conf.org/proceedings/lrec2008/.

[213] Paul Taylor, Alan W. Black, and Richard Caley. Heterogeneous relation graphs as a formalism for representating linguistic information. *Speech Communications*, 33(1-2): 153–174, January 2001. ISSN 0167-6393. doi: 10.1016/S0167-6393(00)00074-1. URL http://dx.doi.org/10.1016/S0167-6393(00)00074-1.

[214] Elke Teich and Mônica Holtz. Scientific registers in contact: An exploration of the lexicogrammatical properties of interdisciplinary discourses. *International Journal of Corpus Linguistics*, 14(4):524–548, 12 2009.

[215] Elke Teich, Silvia Hansen, and Peter Fankhauser. Representing and querying multi-layer corpora. In *Proceedings of the IRCS Workshop on Linguistic Databases*, pages 228–237, Philadelphia, 11-13 December 2001. University of Pennsylvania.

[216] Heike Telljohann, Erhard Hinrichs, and Sandra Kübler. The TüBa-D/Z treebank: Annotating German with a context-free backbone. In *Proceedings of the Fourth International Conference on Language Resources and Evaluation (LREC 2004)*, pages 2229–2235, Lisbon, Portual, 2004.

[217] The DUCC Team. Distributed UIMA cluster computing (0.8.0). Technical report, Apache UIMA, Sep 2013.

[218] The Unicode Consortium. *The Unicode Standard, Version 6.2.0*. The Unicode Consortium, Mountain View, CA, 2012. ISBN 978-1-936213-07-8.

[219] Kristina Toutanova, Dan Klein, Christopher D. Manning, and Yoram Singer. Feature-rich part-of-speech tagging with a cyclic dependency network. In *Proceedings of the 2003 Conference of the North American Chapter of the Association for Computational Linguistics on Human Language Technology - Volume 1*, NAACL '03, pages 173–180, Stroudsburg, PA, USA, 2003. Association for Computational Linguistics. doi: 10.3115/1073445.1073478. URL http://dx.doi.org/10.3115/1073445.1073478.

[220] TreeTagger. A language independent part-of-speech tagger. URL http://www.cis.uni-muenchen.de/~schmid/tools/TreeTagger/ (Last accessed: 2013-09-27).

[221] Reut Tsarfaty. A unified morpho-syntactic scheme of Stanford dependencies. In *Proceedings of the 51st Annual Meeting of the Association for Computational Linguistics (Volume 2: Short Papers)*, pages 578–584, Sofia, Bulgaria, August 2013. Association for Computational Linguistics. URL http://www.aclweb.org/anthology/P13-2103.

[222] TT-NL. Tag set for Dutch used by TreeTagger. URL http://www.cis.uni-muenchen.de/~schmid/tools/TreeTagger/data/dutch-tagset.txt (Last accessed: 2013-10-04).

[223] Dieter Van Uytvanck, Claus Zinn, Daan Broeder, Peter Wittenburg, and Mariano Gardellini. Virtual Language Observatory: The portal to the language resources and technology universe. In Nicoletta Calzolari, Khalid Choukri, Bente Maegaard, Joseph Mariani, Jan Odijk, Stelios Piperidis, Mike Rosner, and Daniel Tapias, editors, *Proceedings of the Seventh International Conference on Language Resources and Evaluation (LREC'10)*, pages 900–903, Valletta, Malta, may 2010. European Language Resources Association (ELRA). ISBN 2-9517408-6-7.

[224] Leonoor Van Der Beek, Gosse Bouma, Rob Malouf, and Gertjan Van Noord. The Alpino Dependency Treebank. In *Computational Linguistics in the Netherlands*, pages 8–22, 2001.

[225] W3C. XML - Extensible Markup Language. Technical report, World Wide Web Consortium (W3C), (http://www.w3.org/XML), 1997.

[226] W3C. XML Schema parts 0, 1 and 2. W3C recommendation, W3C, October 2001. URLs: http://www.w3.org/TR/.

[227] Stephen Wu, Vinod Kaggal, Dmitriy Dligach, James Masanz, Pei Chen, Lee Becker, Wendy Chapman, Guergana Savova, Hongfang Liu, and Christopher Chute. A common type system for clinical natural language processing. *Journal of Biomedical Semantics*, 4(1):1, 2013. ISSN 2041-1480. doi: 10.1186/2041-1480-4-1. URL http://www.jbiomedsem.com/content/4/1/1.

[228] Fei Xia. The part-of-speech tagging guidelines for the Penn Chinese treebank (3.0). Technical report IRCS-00-07, University of Pennsylvania, Oct 2000.

[229] Nianwen Xue. Annotation guidelines for the Chinese proposition bank. Draft, Brandeis University, Feb 2007.

[230] Nianwen Xue, Fei Xia, Shizhe Huang, and Anthony Kroch. The bracketing guidelines for the Penn Chinese treebank (3.0). Technical report IRCS-00-08, University of Pennsylvania, Oct 2000.

[231] F. Yergeau. UTF-8, a transformation format of ISO 10646 (RFC-3629). URL http://tools.ietf.org/html/rfc3629 (Last accessed: 2013-06-18), November 2003.

[232] Seid Muhie Yimam, Iryna Gurevych, Richard Eckart de Castilho, and Chris Biemann. Webanno: A flexible, web-based and visually supported system for distributed annotations. In *Proceedings of the 51st Annual Meeting of the Association for Computational Linguistics: System Demonstrations*, pages 1–6, Sofia, Bulgaria, August 2013. Association for Computational Linguistics. URL http://www.aclweb.org/anthology/P13-4001.

[233] Amir Zeldes, Anke Lüdeling, Julia Ritz, and Christian Chiarcos. ANNIS: a search tool for multi-layer annotated corpora, 2009. URL http://edoc.hu-berlin.de/docviews/abstract.php?id=36996.

[234] Torsten Zesch and Jens Haase. HOO 2012 shared task: UKP Lab system description. In *Proceedings of the Seventh Workshop on Innovative Use of NLP for Building Educational Applications at NAACL-HLT*, pages 302–306, June 2012.

[235] Torsten Zesch, Iryna Gurevych, and Max Mühlhäuser. Analyzing and accessing Wikipedia as a lexical semantic resource. In *Data Structures for Linguistic Resources and Applications*, pages 197–205, Tübingen, Germany, April 2007. Gunter Narr, Tübingen.

[236] Torsten Zesch, Omer Levy, Iryna Gurevych, and Ido Dagan. UKP-BIU: Similarity and entailment metrics for student response analysis. In *Proceedings of the 7th International Workshop on Semantic Evaluation (SemEval 2013), in conjunction with the 2nd Joint Conference on Lexical and Computational Semantics (*SEM 2013)*, volume 2, pages 285–289, Stroudsburg, PA, USA, June 2013. Association for Computational Linguistics. ISBN 978-1-937284-49-7.

[237] Jun Zhao, Jose Manuel Gomez-Perez, Khalid Belhajjame, Graham Klyne, Esteban Garcia-Cuesta, Aleix Garrido, Kristina Hettne, Marco Roos, David De Roure, and Carole Goble. Why workflows break – understanding and combating decay in Taverna workflows. In *8th IEEE International Conference on eScience (eScience 2012)*, pages 1–9. IEEE, 2012.