Large-scale Information Extraction from Neuroscientific Literature

Marco Antognini

Spring 2015

Master Semester Project under the supervision of
Jean-Cédric Chappelier & Renaud Richardet
Artificial Intelligence Laboratory LIA - EPFL

Abstract

This report presents a new, clever strategy for automatic remote dependencies handling in Sherlok, an open source, large-scale, text mining, RESTful service based on UIMA technologies like bluima, which focuses on neuroscientific data extraction to help scientists at the Blue Brain Project collect information such as connections between brain regions from the vast available literature. The presented strategy is based on a thorough analysis of Sherlok's needs. Additionally, we suggest a general direction for further development of Sherlok and bluima.

Contents

1 Introduction

The neuroscientists at the Blue Brain Project [1], in their quest to model the human brain, have to understand the interactions between the different components of the brain such as cells or brain regions and connect their effects with other parts of the brain or other organs. However, since the amount of information is colossal and is obviously not memorisable by individual human being or even by a team of researchers, tools need to be created in order to extract the relevant data from the scientific literature in a precise and fast manner.

Sherlok [2] was created to solve this large-scale text mining problem. Based on a Representational State Transfer (REST) API, Sherlok is a service capable of extracting information relevant to specific situation directly from a given text. It is designed to be both user- and developer-friendly and flexible so that it can analyse text via standard REST requests.

Currently, the main dependency of Sherlok is, but is not restricted to, bluima [3] [4], which is an open source project developed at the Blue Brain Project and consists of a large collection of UIMA-based *engines* [6] (also called *annotators*) that generate metadata from the input text. We will focus on those specific engines for this project.

Generated metadata consists of a collection of *annotations*. Each annotation has a begin and an end position in the raw text, a type identifier and in some cases extra attributes. It ranges from tags as simple as sentences boundaries to more complex notions such as brain regions.

For example, annotating the following sentence for brain regions, generates the following annotations (for simplification, part of speech (PoS) tags are not shown here). Additionally, Sherlok reports brain regions ① and ② as a co-occurrence.

Substantial numbers of tyrosine hydroxylase-immunoreactive cells in the $\underbrace{\text{dorsal raphe nucleus}}_{\text{Brain Region ①}}$ were found to project to the $\underbrace{\text{nucleus accumbens}}_{\text{Brain Region ②}}$.

While each engine works on its own to generate metadata, Sherlok lets the user combines them in *pipelines* where the output of one engine is used as the input for the next one. And in order to reuse engines in different pipelines, Sherlok separates their definitions: engines are first grouped in bundles, which make the bridge between Sherlok's scripting system and the UIMA engines written in Java so that they can be used by several pipelines.

In addition to basic configuration parameters such as arbitrary strings or integers, the algorithms used in the different engines can rely on external resources which can be as simple as a file containing a list of countries, but could also consist of large trained models (e.g. to detect part of speech) or directories of configuration files. The main goal of this project was to decouple those algorithms from their resources in bluima in order to add flexibility to the installation, update and usage of those resources in Sherlok.

Initially, bluima relied on a system-wide setting to locate its resources. However, since we wanted to make Sherlok run on as many systems as possible – which should include a server architecture where a dedicated machine is in charge of storage – we couldn't rely on this property any longer. Hence, the first and main task of this project was to devise a system capable of working without hard coded paths.

This project was threefold:

- the first and main task was to devise a system capable of working without hard coded paths;
- a second task was the design of a clever system for installing and loading resources on demand (i.e. when loading a pipeline);
- the third and final task was to port a full brain regions NER and connections pipeline from bluima to Sherlok, which includes sentence detection, part of speech tagging, species recognition, measures and units extraction and more [4].

The remainder of the report is constructed as follows. Section 2 further introduces Sherlok and its working. Section 3 then presents bluima and sketches how UIMA pipelines work. Section 4 presents the central work of this project, the new packaging strategy and how Sherlok can install remote resources automatically for the users, that is. Finally, Section 5 summarises the work that was done in this project and proposes some possible future developments in Sherlok and bluima.

2 Sherlok

Sherlok is an open source service for text information extraction created at the Blue Brain Project to extend and popularise bluima engines. However, Sherlok is not restricted to neuroscience and as such, does not require any prior neuroscientific knowledge.

2.1 REST-Based Service

The service, provided through a REST API, can be installed in-house by system administrators to match the specific needs of researchers. From the user perspective, only a basic understanding of the Hypertext Transfer Protocol (HTTP) is required to build queries used to communicate with Sherlok.

While its implementation is written in Java, any language or tool with basic socket support can communicate with Sherlok and benefits from its capabilities without going through the frequent required hassle to build wrappers to communicate between Java and another language. Additionally, as a proof of concept, Sherlok provides a JavaScript, Java and Python client API to emit the standard REST queries in those languages. And due to the nature of REST and HTTP protocols they can easily be extended to support more languages.

From an end-user point of view, the HTTP request

GET /annotate/<pipeline>?text=<input>

is all there is to know to use a given pipeline to process some text: nothing is installed on the user side; instead everything is directly available on Sherlok's server without investing in powerful personal computers. And if the existing pipelines don't fit the user's needs, then a new pipeline can be submitted to Sherlok which will automatically install the required dependencies, would they be Java packages or regular files needed at runtime. Additionally, Sherlok comes with a basic web editor to easily install new pipelines, edit them and process text without manually writing any REST queries.

From a system administrator point of view, Sherlok is no less flexible or harder to use. The installation procedure is as simple as extracting an archive and launching an executable. And because Sherlok runs on the Java Virtual Machine (JVM), any system with a proper Java Runtime Environment 7 (JRE7) is able to run Sherlok. The memory and disk requirement depends on the pipelines that the users will run. However, the system administrator doesn't have to pay resources for pipelines that are not used: Sherlok easily allows obsolete pipelines or resources to be deleted.

Were one server not capable of providing enough computational power to handle all users' requests, several instance of Sherlok could be run in a *slave* mode to balance the workload on several machines.

Moreover, if the system administrator wanted to restrict the installation or modification of pipelines, he could run Sherlok in *sealed* mode to freeze the configuration of the service. While Sherlok doesn't have any particular Access

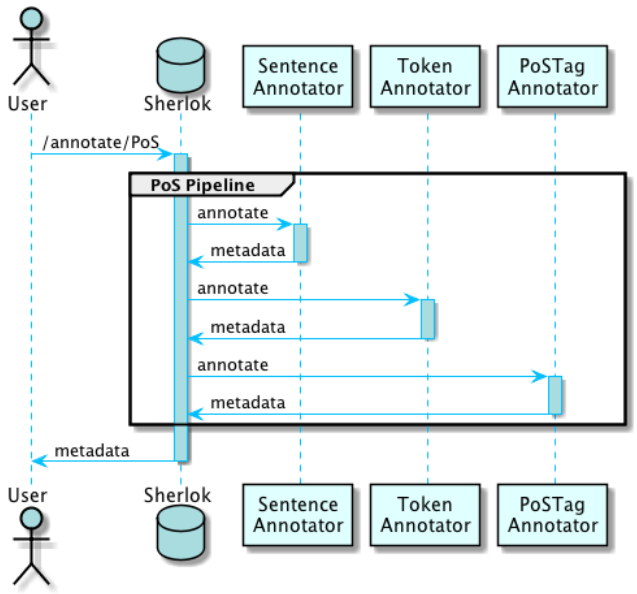

Figure 1: Part of Speech Pipeline

Control List (ACL) mechanism to support fine-grained access control and restrict users permissions on a case by case basis, custom ACL system can easily be built on top of Sherlok: a simple proxy program can integrate restricted access using for example Unix accounts or a Lightweight Directory Access Protocol (LDAP) server to filter requests before Sherlok can process them.

Figure 1 depicts how a REST query is handled by Sherlok from a very high level point of view.

2.2 UIMA-Based Configurable Service

Internally, Sherlok processes the input text through Unstructured Information Management Architecture (UIMA) [6] engines or more generally through a Rule-based Text Annotation (RUTA) scripts [11] [12]. Both RUTA and UIMA are projects of the Apache Software Foundation [16] focusing in analysing large volumes of unstructured information in order to discover knowledge that is relevant to an end user. While Sherlok focuses on the Java Frameworks for UIMA, it could be extended in the future to also support the C++ or Scaleout Frameworks.

In order to let users, who are not Java developers, combine engines together, Sherlok has a Java agnostic configuration system that separate the definition of engines from their usage in pipelines.

Those configuration files make use of two concepts:

Bundles The user lists the UIMA engines that he wants to use in *bundles* by

```
"DocumentAnnotation" : [
  { "begin" : 0,    "end" : 328,  "language" : "en" }
],
"Sentence" : [
  { "begin" : 0,    "end" : 135,
    "componentId" :
        "de.julielab.types.OpenNLPSentenceDetector" },
  { "begin" : 136,  "end" : 32
    "componentId" :
        "de.julielab.types.OpenNLPSentenceDetector" }
]
```

Listing 1: Excerpt of JSON output for sentence detection

specifying the Java package dependencies, the required resource files and configure the different parameters of the engines. To simplify installing resources, Sherlok supports download at execution time of Git repositories or HTTP(S) URLs, but also lets the user specify local path on the server if needed.

Pipelines The user writes the RUTA script which can be as simple as chaining engines from the bundles configuration files in *pipelines*. The whole features set of RUTA is available to the user so that simple scripts don't require any Java implementation from the user. Similarly to bundles, Sherlok allow downloading resources for pipeline automatically.

By default Sherlok comes with pipelines to extract information about brain regions, part of speech and much more. The engines used in those pipelines come from bluima, which is presented in section 3. Listings 4, 5 and 6 give some concrete examples of bundles and pipelines.

The annotations produced by a pipeline are returned from the server in JSON format [5]. For example, listing 1 highlights the annotations produced by the BLUIMA.SENTENCE pipeline when processing "*Terminologies which lack semantic connectivity hamper the effective search in biomedical fact databases and document retrieval systems. We here focus on the integration of two such isolated resources, the term lists from the protein fact database UNIPROT and the indexing vocabulary MESH from the bibliographic database MEDLINE.*"

In this listing are displayed three types of annotations: one DOCUMENTANNOTATION, which provides insight on the language of the processed document, and two SENTENCE, each of which have a general COMPONENTID representing which NLP component has been used to derive the annotation. Additionally, all three annotations have information about the corresponding range of the text that they cover.

For quality purpose, Sherlok also allows the user to embed tests for pipelines that can be run through the GET /test/<pipeline> query in order to ensure that pipelines continue to work properly after updating Sherlok or the engines' dependencies.

3 Bluima, UIMA and RUTA

Bluima is an open-source collection of UIMA readers [6] augmented by multiple engines developed at the Blue Brain Project [1] to perform large-scale information extraction from the neuroscientific literature. It includes for example a species engine based on the Linnaeus library [19], or the OpenNLP toolkit [17] to perform PoS tagging, sentence splitting and more. Table 1 lists the major subprojects used by bluima engines.

OpenNLP [17]	A machine learning based toolkit for the processing of natural language text.
Linnaeus [19]	A general-purpose dictionary matching software, capable of processing multiple types of document formats in the biomedical domain.
BANNER [20]	A named entity recognition system, primarily intended for biomedical text.
Dragon Toolkit [21]	A Java-based development package for academic use in information retrieval and text mining.
BioAdi [22]	BioAdi identifies the mapping between Short Form and Long Form terms in paper.
langdetect [23]	A language detection library implemented in plain Java.
JWI [24]	A Java library for interfacing with Wordnet.
jSRE [25]	A Java tool for Relation Extraction.
JULES [26]	A collection of UIMA wrappers for OpenNLP.
OSCAR4 [27]	An open source extensible system for the automated annotation of chemistry in scientific articles.

Table 1: bluima aggregated projects

UIMA-based applications are structured into components called *analysis engines*, or *annotators*, which produce metadata in form of annotations. The implementation of such engines can be either written in C++ or in Java, however C++ development seems rather marginal compared to the amount of available libraries and tool for UIMA in Java, for which several implementation strategies have been devised over the years. The most modern and popular API used to build components is *uimaFIT* [7] [9]: it aims to simplify the archaic XML description used in earlier UIMA APIs to configure (C++ or Java) engines by writing both the implementation and configuration of engines' parameters directly in Java. While Sherlok is compatible with any UIMA-compatible technology, we focus on the uimaFIT approach in what follows.

At runtime, an engine has mainly two actions: initialisation and processing. Annotations produced by engines are stored in a Common Analysis System (CAS) [10] general purpose data structure, which allows the representation of objects with single-inheritance hierarchy, properties and values. And in a pipeline of engines, a single CAS instance is passed from one engine to the next so that the data from previous engines can be used to filter data, augment existing annotations or produce new ones.

Figure 2 depicts how uimaFIT-based applications handle pipelines such as PoS tagging: first, the text to be processed is loaded into a CAS instance, then engines are created through uimaFIT factory system with the appropriate configuration settings and finally the pipeline of engines is run.

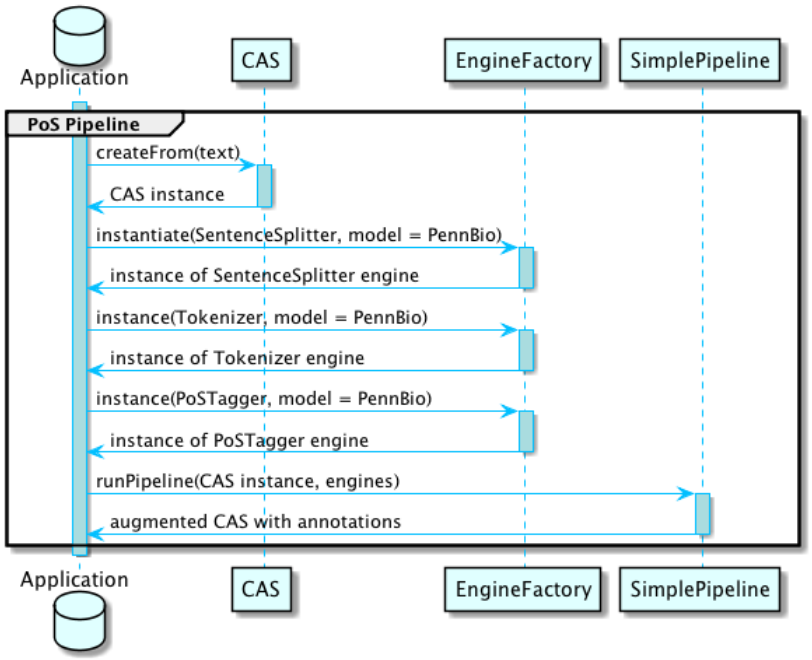

Figure 2: Part of Speech tagging uimaFIT pipeline

While uimaFIT is very convenient when writing an UIMA application that is entirely written in Java, it becomes much more complex when we want to include a flexible external configuration as we do with Sherlok. Moreover, and especially for simple pipelines, sometimes implementing a Java engine requires more boilerplate code than the actual algorithm.

To workaround this, an application can use the RUTA language [11] [12], which is an imperative rule-based language extended with scripting elements that can also be used to define pipelines. To illustrate this, listing 2 shows a very simple script that annotates "dog" as an "Animal" and listing 3 shows how RUTA can be used to run the PoS pipeline from figure 2. For this last example,

```
DECLARE Animal;
W{REGEXP("dog") -> MARK(Animal)};
```

Listing 2: A basic RUTA script

```
ENGINE SentenceAnnotator;
ENGINE TokenAnnotator;
ENGINE PosTagAnnotator;

Document{-> EXEC(SentenceAnnotator)};
Document{-> EXEC(TokenAnnotator)};
Document{-> EXEC(PosTagAnnotator)};
```

Listing 3: PoS pipeline written in RUTA

unlike with uimaFIT, the XML description of engines' parameters need to be available at runtime as it is not embedded in the script itself.

Fortunately, uimaFIT provides tools to generate XML descriptions for classic engines so that we can use RUTA to generate pipelines automatically for the user from a simpler description of the engines' parameters than raw XML as follows:

1. the user defines the engines with their parameters inside bundles;
2. then he defines the order of engines in a given pipeline;
3. when the user wants to annotate a given text with the pipeline, Sherlok converts the engines, with their configured parameters, used by the pipeline into XML descriptions;
4. Sherlok generates a RUTA script that uses the engine descriptions in the order of the pipeline;
5. and finally, Sherlok runs the RUTA script and return the annotations back to the user.

Additionally, Sherlok allows users to use the full RUTA language in pipelines to create even more powerful tools. And in order to make configuration more flexible, as we will see in section 4.4, the user has the possibility to use configuration variables in bundles or pipelines to download automatically remote resources.

4 Packaging Strategy

Initially all resources used by bluima's engines, such as models trained on the GENIA corpus [28] to tag part of speech elements, were accessed via a system wide path variable. In this section we will discuss the adopted strategy to improve this system and make bluima "path-agnostic" in order to improve its integration in Sherlok. In the Appendix A we show a few other alternatives that were also considered during this project.

4.1 Initial Situation

Both bluima and Sherlok build system are based on Apache Maven [13], which allows developers to define how a Java project should be compiled, packaged, tested and much more. One important feature of Maven is its ability to download dependencies from centralised repositories where developers can publish their releases. At the beginning of this project, however, bluima was not yet available in Maven central which meant that to use it one had to clone its Git repository, compile and install it.

Yet, publishing bluima to Maven Central would not suffice to solve a bigger issue. Bluima widely uses resource files in its engines and therefore needs to access those files at runtime. Originally, it used a system global variable (`BLUIMA_HOME`) as many popular tools do nowadays (e.g. Java with its `JAVA_HOME`) to locate specific files. Thus, using Maven to download and install Java archives (JAR) is not enough to locate those files later at runtime: the variable must be set at a system-wide level by the user itself.

This made it clear that we needed a better engines-resources model or the integration of bluima in Sherlok would imply that system administrators would have to manually install and configure bluima, which in itself would defeat the whole purpose of Sherlok to make users' life better and automate dependencies installation.

4.2 Proposed Solutions

There exist, as always, many methods to address this issue. But for this project, in addition to ease the integration of bluima in other tools, we wanted to keep the refactoring and maintenance cost low, allow resources to refer to one-another, make it possible to group files together in *batches*, avoid runtime performance penalties wherever possible and remove as little flexibility from the users as possible.

4.2.1 Refactoring and Maintenance

The first idea that we explored was to package resources inside JAR files and use Maven to distribute them similarly to what DKPro [14] [15] does but in a less complex manner for the users.

In a nutshell, packaging resources within JAR files has two important drawbacks. Firstly, updating a resource file implies repackaging and redeploying the new version to, for example, Maven Central, which can be quite a burden. One could argue that this is an administrative task that is not frequent and therefore not significantly enough to discard this strategy. However, the second issue is of much more concern.

As detailed in appendix A, this packaging strategy implies more refactoring than it seems at first sight: in order to be able to use an engine with different resources (e.g. one might prefer to use a probabilistic tokenizer trained on the GENIA corpus to process some inputs, but use a model trained on a different corpus for some other inputs) and not hardcode them, we need a *proxy* to select from an identifier the appropriate files at runtime.

In appendix A, we illustrate this with a technique using as the identifier the name of the class that "owns" the desired resource file. Although the fact that the implementation of the proxy we designed is based on the Java Reflection API is not related to the amount of refactoring needed to adapt bluima to this packaging design, it significantly adds some complexity to the overall software and therefore will increase its maintenance cost.

The Java API to access JAR content is based on read-only access because, among other things, those archives can be signed and thus modifying their content would threaten the integrity of the signature or corrupt the archive.

The core refactoring issue with this design is that instead of working with File objects or paths as the initial implementation of bluima engines does, we would have to use InputStream in order to access the resources files because of how Java JAR system was designed. And this can be really problematic for some engines. For example, the LinnaeusAnnotator, which wraps the external Linnaeus library [18], uses an ArgParser internally that cannot be created from a data stream but absolutely requires a path to a configuration file.

Of course, some workarounds could be elaborated, such as copying the content of the file from the JAR file to a temporary file and then use this file's path, but only on a case by case basis and that those would most probably induce performance penalties.

4.2.2 Resource Interdependency

The LinnaeusAnnotator also illustrates another feature that engines might need: the ability to reference a resource file from another resource file via relative paths. With DKPro-like solutions, the usage of stream to read resources content makes it much harder to allow such behaviour.

Workarounds for this specific issue usually imply modifying the engines and their underlying implementation to work differently. In the case of LinnaeusAnnotator this would have meant to completely rewrite the Linnaeus library since it is a closed-source utility, which is clearly out of the scope of this three-month project.

4.2.3 Batch Of Files

Like with LinnaeusAnnotator, the BrainRegionPipes use a batch of files: a collection of configuration files used together to set up an engine or its dependencies. For the BrainRegionPipes the problem is easier since no file makes a reference to another one. Nevertheless, it would be very verbose and painful to denote each and every of the 30 files one by one in the configuration of the engine. Hence, considering a batch of files simply as a collection of files without a specific naming structure is not optimal. Depending on whether or not we base the resource packaging system on a system similar to DKPro, the consequences differ:

- Were we to use a packaging strategy based on JAR files, we could introduce an arbitrary naming structure for the identifiers used to access the actual resource through the proxy system and have an *aggregator proxy* that generates multiple data streams to each and every individual resources from a single root identifier. While this solution seems simple on the paper it actually requires as much refactoring as using one identifier by resource.

  Additionally, it doesn't solve the interdependency between resources as illustrated above with LinnaeusAnnotator.

- If we apply the idea of names having an arbitrary but constant structure to the filenames directly instead of the identifiers used by a proxy, we could group the resource files together in an archive (e.g. a zip file) that would get extracted at runtime into a temporary directory.

  With this approach we have at least two significant advantages over the JAR-based design without sub-archives. Firstly, it only requires a minimal refactoring of the current Java code to extract and use the data as the implementation is based on Files or Strings representing paths. Secondly, it allows files to refer to one another through relative paths.

  However, this would imply adding a relatively complex version checking system in bluima and/or Sherlok: in order to not extract data from an archive when it is already available to prevent degrading performance too much, we would have to check if the data was already extracted and if that is the case that it is actually correct. This can be quite complex when, for example, upgrading resources. And this is without mentioning having several pipelines using the same engine but with different version of its resources, nor dealing with multiple concurrent annotation instances. Finally, if we take the example of Sherlok running on several servers, we introduce an issue of data accessibility and replication for cache purposes.

4.2.4 Flexibility

With all the above issues, the JAR-based packaging design seems rather inconvenient to use, mainly because it involves a lot of refactoring. Moreover it adds some complexity to the implementation. Some engines behave differently depending on the file type of some of their resources: for example, the

SuffixSensitiveGISModelReader utility reads the content of its input resource differently depending if the file extension is "`.bin`" or "`.txt`". When using InputStream instead of File the extension is lost and therefore need to be tracked with an additional parameter.

In addition to all the above issues, we could argue that we actually remove flexibility from the users by forcing them to use a Java system to archive and manage resources despite the fact that the system using bluima could be Java agnostic from the final user point of view. It could actually be quite painful for someone who is using a Source Control Management (SCM) tool, such as Git, to manage two distinct files structures: one that is convenient to the user and one for our JAR-based archiving system.

Moreover, if we take the example of batch of files, at the end of the day, we have extracted some data that the user had to package initially in order to make the installation process easy but incredibly complexify the implementation.

4.2.5 Adopted Solution

We argue that the installation process for runtime resources could be much simpler than packaging resources into JAR files in the first place, potentially aggregating batch of files together in sub-archives, then accessing those files via a proxy and an extra layer of resource identifiers, potentially extracting sub-archives into a temporary directory, and by this mean forcing the user to use a specific versioning system as DKPro-like systems do.

The JAR-based system that we present in Appendix A can, if needed, add an additional runtime safety to ensure that resources are present. However, we argue that this additional security layer is not really necessary since it is rather trivial to detect a missing file from an engine at runtime and propagate an error to the user, but also because this kind of situation should not occur at all once the pipelines have been tested. Moreover, we can directly make some basic checks in Sherlok that are in practice sufficient. For these reasons we don't consider it as a critical downside.

From bluima perspective, the user could simply install the configuration files in a directory of his choosing and configure the engines with paths to the actual resources without using any system-wide variable. Withal, since this simpler approach does not enforce any versioning mechanism at all, the user is free to implement any system that matches his needs. Following from this fact, Sherlok can add an extra layer, directly integrated in the bundle and pipeline configuration files, to manage resources easily for the end users. Such system can, but is not restricted to, be based on Git: a user can define a (batch of) resource(s) by a Git repository and an optional tag, branch or commit SHA and Sherlok then can automatically download and install the specific version of the resources.

This last solution allies the flexibility of using Java File objects, and therefore involves only little refactoring of bluima engines, with the power of external, well-establish and powerful SCM tools in addition to avoid the limitation of having to rely solely on InputStream to access resources. This is why it

was selected and implemented in this project.

We present in table 2 a quick summary of the pros and cons of all the techniques we have explored for this project. The four "*proxy-based*" versions are detailed in appendix A. The first column reports the workload needed to implement and use the corresponding approaches in bluima. The second one indicates if the code can be constructed in a concise way (DRY standing for *Don't Repeat Yourself*). The flexibility column tells how easy it is for users to integrate their resources. The fourth one represents the amount of work needed to propagate a new version of some resources to the packaging system. And finally the last one states whether or not the system integrates a runtime safety net.

Strategy	Refactoring-Friendly	DRY-code	Flexibility	Repackaging Cost	Runtime Safety
DKPro-Like [14]	✗	✓	(✓)	✗	✗
Proxy-Based A	✗	✓	✗	✗	✓
Proxy-Based B	✗	✓	✗	✓	(✓)
Proxy-Based C	✗	✗	✗	✗	✓
Proxy-Based D	✗	✓	✗	✓	✗
Adopted Solution	✓	✓✓	✓✓	✓	✗

Table 2: Packaging Strategies Comparison

4.3 Impact on bluima

One of the important aspect of the adopted solution is that it is actually easy to implement: bluima only need some minor refactoring and it can be extended to other project easily, regardless of their implementation language or tools.

Because we now want to work only with generic paths, the refactoring mostly consists in adding configuration parameters to engines to handle custom resource locations. For example, the `modelFile` parameter of SentenceAnnotator, which represents the model to use for detecting sentences, should be capable of loading a file from wherever on the disk. In practice this change is trivial and since this only adds flexibility, we don't see any downside to this technique.

However, this implies that a marginal part of the current code base of bluima needs to get upgraded, deprecated or even removed. For example, the

OpenNlpHelper class is a utility to load NLP engines, such as SentenceAnnotator, and configure them with the models trained on the PennBio corpus. It cannot be used as-is because this utility uses a fixed path, based on the global variable `BLUIMA_HOME`, to locate model resources. For this specific case, it is easier to simply deprecate it for mainly two reasons: *a*) using the NLP annotators directly is straightforward; *b*) and more importantly, this factory is not compatible with RUTA and Sherlok scripting system.

Additionally, unit tests need to be refactored to take into account that using the `BLUIMA_HOME` environment variable is no longer reliable. Fortunately, this is rather trivial for bluima since its build system is Maven: unit tests can be configured through the maven-surefire-plugin at compile time, without any user interaction, to set a Java property that represents the root directory of all bluima resources. Then, the refactoring of tests is as simple as using the newly introduced Java property instead of a system-wide variable.

Finally, because engines are now independent of the location of their resource files, we can move the resources into a separate Git repository and include it in bluima with a Git submodule. As we will see in the section 4.4, this has the nice property of allowing Sherlok to download only those resources at runtime, and thus spare the effort of downloading the full bluima code base which weighs more than 700MB (with its history) on Sherlok's server.

4.4 Configuration in Sherlok

Now that we have system that allows resources to be located anywhere on the filesystem and used at runtime, we can go further and propose a flexible integration system for resources in Sherlok.

We can use the fact that resources are identified when configuring engines to let Sherlok download them automatically, similarly to regular Maven dependencies for the actual engine implementation. To do that, we add the ability to define and use *configuration variables* in bundles and pipelines that will, after the resources have been downloaded to an internal directory of Sherlok, be substituted by the actual path to the corresponding resources.

For example, listing 4 shows how the SentenceAnnotator engine can be defined in a bundle with *a*) its Java dependencies using Maven artefacts (here the only dependency is bluima_opennlp which contains the Java class ch.epfl.bbp.uima.ae.SentenceAnnotator) and, *b*) its runtime resources (here the trained model `SentDetectGenia.bin.gz` from the master branch of bluima_resources Git repository).

When defining a pipeline in Sherlok that uses RUTA scripting elements, users might want also to fetch runtime resources as well. Therefore, we added the same ability to define configuration variables inside pipelines. Moreover, Sherlok is capable of downloading remote resources over HTTP, as shown in listing 5 where a list of countries is downloaded when the pipeline is used. The only particularity with resources that are used directly in RUTA scripts is that they need to be in a special mode because the underlying path needs to be

```
{
  "name": "bluima.sentence",
  "domain": "examples",
  "version": "1.0.1",
  "dependencies": [
    {
      "value": "ch.epfl.bbp.nlp:bluima_opennlp:1.0.1"
    }
  ],
  "config": {
    "bluima": {
      "type": "git",
      "url":
        "https://github.com/BlueBrain/bluima_resources.git",
      "ref": "master"
    }
  },
  "engines": [
    {
      "name": "SentenceAnnotator",
      "class": "ch.epfl.bbp.uima.ae.SentenceAnnotator",
      "parameters": {
        "modelFile": [
          "$bluima/opennlp/sentence/SentDetectGenia.bin.gz"
        ]
      }
    }
  ]
}
```

Listing 4: Configuration of SentenceAnnotator in a Sherlok bundle

```
{
  "name": "countries",
  "version": "1",
  "description": "Example that annotates countries",
  "domain": "examples",
  "config": {
    "countries": {
      "type": "http",
      "url": "https://example.com/countries.txt",
      "mode": "ruta"
    }
  },
  "script": [
    "WORDLIST CountriesList = '$countries';",
    "DECLARE Country;",
    "Document{-> MARKFAST(Country, CountriesList)};"
  ]
}
```

Listing 5: Configuration of the COUNTRIES pipeline

computed slightly differently. This is expressed by adding `"mode" = "ruta"` to a configuration variable definition.

Yet, downloading remote resources can be costly and should only be done when the resources are needed and not repeated unless needed. This is why Sherlok will cache resources once they have been fetched the first time they are used. Conversely, keeping resources that are not used can be costly for the system administrators. Hence why they can be removed through REST calls, such as DELETE /clean/runtime_resources that will remove any downloaded resources.

Additionally to Git and HTTP configuration variables, Sherlok also supports basic text variable and is designed to be easily extended to support more type of remote resources, such as SVN or Mercurial repositories, remote files over SSH or any other communication protocol.

To support a new protocol, Sherlok code base needs to be extended as follows:

1. a new class implementing the CONFIGVARIABLE Java interface and supporting the new protocol needs to be created;
2. the CONFIGVARIABLEFACTORY.FACTORY method needs to be augmented to create the new kind of variable when appropriate;
3. and, optionally, the CONFIGVARIABLEFACTORY.CLEANERFACTORY method can be extended to support cleaning resources of the new kind.

To conclude this section, figure 3 depicts how bundles and pipelines are combined together to generate the underlying RUTA scripts that Sherlok will use to annotate text.

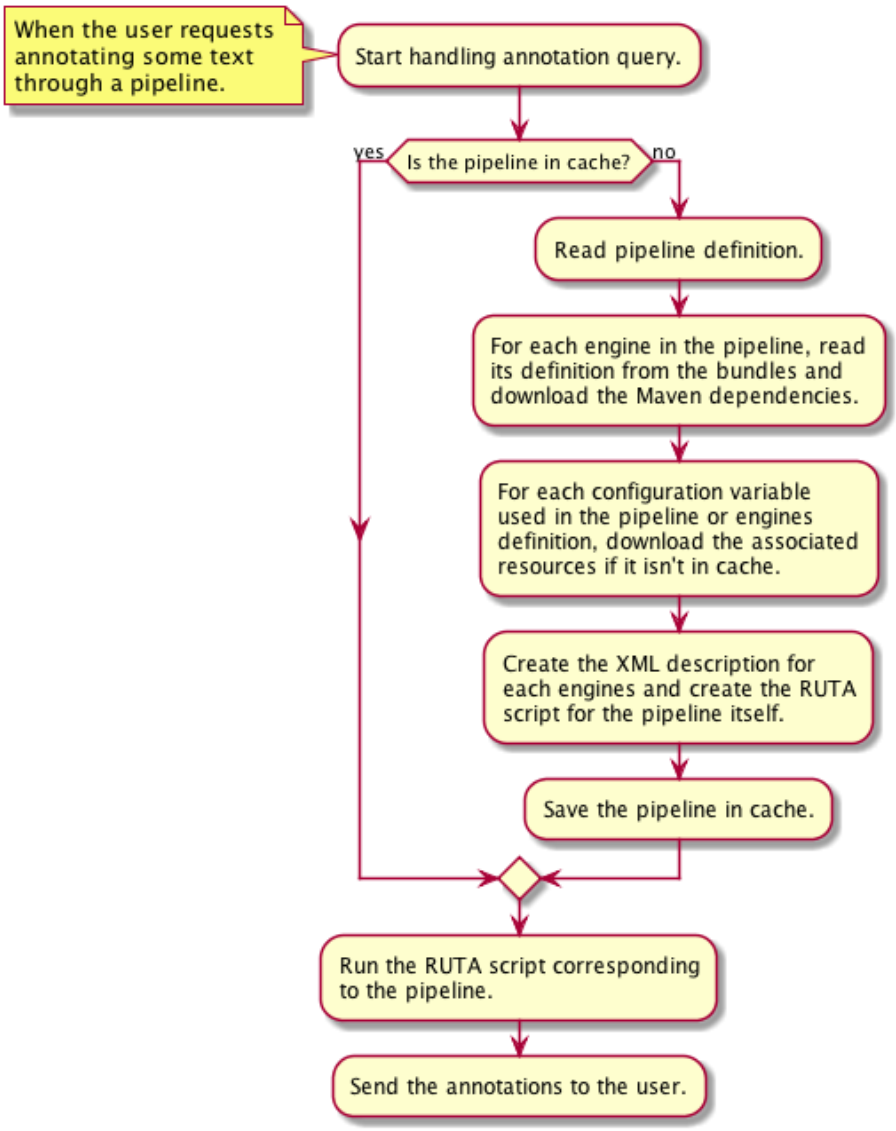

Figure 3: Sherlok pipeline loading procedure

4.5 Possible Extensions

The implementation of the previous concept in this project proves that this strategy works well for Sherlok. But yet users might want more features, such as more remote protocol support (SVN, hg, SSH, ...) as previously mentioned. Here we list a few other features that could be interesting to implement.

Archive Extraction When working with batch of files, it can be convenient to download an archive and extract it automatically. This could be easily be added to Sherlok by adding a parameter for the HTTP configuration variable.

Redownload If Newer Being able to fetch remote files, especially with HTTP protocol, is very useful. Currently, once resources have been downloaded and cached by Sherlok they won't be redownloaded until they are cleaned. Therefore, if the remote resource is updated, the system administrator has to clean it manually to get the update. However, this could be easily be improved by supporting automatic update of HTTP resources by using the If-Modified-Since request-header field.

Pull Git Branch Sherlok currently supports selecting specific commit, branch or tag in configuration variable when working with Git repositories. But similarly to HTTP resources, once the data has been fetched and cached it won't be updated until it has been cleaned. It would be convenient to get automatic update from upstream. This could either be simply implemented by pulling the new history at periodic intervals, or by using web hooks or equivalent techniques. But Sherlok would have to be very careful and do it only when the resources are not in use by any pipeline or it would risk corrupting running engines.

Improve Download Strategy While the current implementation works well, there are a few area that could benefit from optimisation. For example, naively cloning the whole Git repository with its complete history only to checkout a specific commit or tag is really under-efficient. Instead, for this specific case, downloading only the most recent history from the server is much faster. This could be archived with the "`--depth 1`" option of Git command line.

```
{
  "name": "bluima.regions_jsre",
  "version": "1.0.1",
  "description": "annotate brain regions",
  "domain": "bluima",
  "script": [
    "ENGINE SentenceAnnotator:1.0.1;",
    "ENGINE TokenAnnotator:1.0.1;",
    "ENGINE PosTagAnnotator:1.0.1;",
    "ENGINE BlueBioLemmatizer:1.0.1;",
    "ENGINE MeasureRegexAnnotator:1.0.1;",
    "ENGINE PruneMeasuresAnnotator:1.0.1;",
    "ENGINE LinnaeusAnnotator:1.0.1;",
    "ENGINE BrainRegionAnnotator:1.0.1;",
    "ENGINE KeepLargestBrainRegionAnnotator:1.0.1;",
    "ENGINE ExtractSameBrainRegionCoocurrences:1.0.1;",
    "ENGINE JsreBrainRegionFilterAnnotator:1.0.1;",
    "ENGINE KeepLargestCooccurrenceAnnotator:1.0.1;"
  ]
}
```

Listing 6: jSRE brain region pipeline

5 Conclusion & Future Development

With this project we have shown that Sherlok can be used to write pipelines for general NLP processing. For example, we were able to create a simple pipeline to detect sentences, but more conclusively we also have created a full brain region detection pipeline, based on jSRE [25], that is displayed in listing 6. Furthermore, we have derived this pipeline into two variants: one using a top-down approach, the other using a rule-based approach.

A fourth technique to detect brain regions is based on ConceptMapper from UIMA Addons and Sandbox [8]. ConceptMapper is defined as follows in its documentation:

> [It] is a highly configurable, high performance dictionary lookup tool, implemented as a UIMA component. Using one of several matching algorithms, it maps entries in a dictionary onto input documents, producing UIMA annotations.

However, it is not directly compatible with Sherlok bundle and pipeline system as it is already in itself an abstraction over UIMA components. Nevertheless, we believe that some mechanism could be set up to accommodate the integration of this powerful tool into Sherlok.

The challenge would be to make users able to express in bundles and pipeline descriptions their intention to use ConceptMapper, which basically takes as

input a XML file containing essentially: *a*) the configuration parameters description; *b*) the configuration parameters values; *c*) and the UIMA type system for annotations. The first point could be abstracted with a uimaFIT engine written in Java and the second correspond to the configuration of such engine in a bundle. The third one could be extrapolated from Sherlok's knowledge of the different engines used in a pipeline or provided by the user.

However, the tricky part is that one of the configuration parameters of ConceptMapper is a XML description of a tokenizer to interpret data from the input dictionary. Since this description can be generated from a uimaFIT engine, one could devise a mechanism in Sherlok to automatically generate it from a selected tokenizer engine with some special syntax in bundle configuration.

On a different topic for improvement, and independently of the extensions mentioned in section 4.5, one could make Sherlok capable of downloading the text that the user wants to annotate through common protocol such as HTTP or SSH: instead of using GET /annotate/<pipeline>?text=<input>, we could imagine that users would specify the remote URL where the text is hosted with, for example:

GET /annotate/<pipeline>?ssh=<url>

Additionally, Sherlok and bluima could be extended to fully support input in different formats, such as PDF or HTML. This is possible by using special engines, called text adaptor, at the beginning of the pipelines that would extract the actual text from the input file and pull out the interesting metadata for the user. Those text adaptors could extract more than just the raw text. For example, a PDF extractor could include information about text emphasis, whether a word is part of a section title, if a proper name is an author of the current document or replace images by a text description.

Appendices

A Proxy-Based Packaging Strategy

The initial idea was to base the system on Apache Maven [13]. However, while implementing the strategy described below we realised that it was not as flexible as predicted. A better alternative is presented in section 4. This appendix describes our first approach.

One of the first objectives of this project was the disassociation of algorithms (engines) and their different resources (model files, configuration files) and devise a packaging system, based on Apache Maven, that would be flexible while being simple and convenient to use and let the user define pipelines using specific engines and models.

A.1 Packaging

With Maven, the main idea is to store a set of resources, such as dataset trained on a specific corpus of texts, in a JAR file and access them in read-only mode at runtime. And after the packaging of resources is done, the JARs are published on a public repository such as Maven Central. Then package dependencies can be used to ensure that some files are available at runtime. And, ideally, package versions should be used to let users update resources independently of the algorithms.

Tools like DKPro [14] already provide method to ship resources alongside algorithms. However, we decided to use a different system because DKPro is rather complex, although somewhat flexible, and because it heavily relies on ANT scripting build system to export data into JAR files, which is not straightforwardly integrable into Maven build system.

It might not be clearly apparent for someone not used to work with Java JAR-based system, but with this approach we cannot modify the resource files at runtime since they have been packaged, and this for mainly two reasons. Firstly, the Java API doesn't provide standard tools to modify such archives. And secondly, if the JAR file were signed (which is the standard procedure) then modifying its content would actually break the signature and therefore corrupt the archive.

Additionally, the API to access resources inside JAR files is not based on Java File objects. Instead algorithms have to use InputStream objects through ClassLoader objects to access the content of a given file. We will discuss how the current code need to be refactored in section A.2.

Here we describe three alternatives that we used to contrast alternative strengths and weaknesses before concluding with a fourth variant that should match our needs. Below, ada and bob denote two variants of a kind of resource that are compatible with a common algorithm package, denoted algo. The using* packages can be thought of as pipelines in the context of bluima/Sherlok.

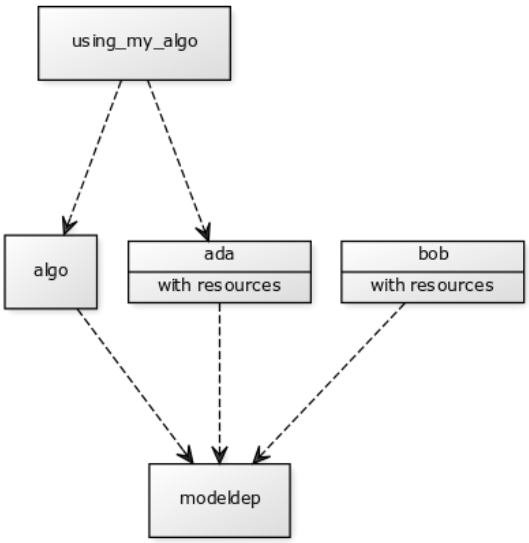

Figure 4: Proxy-based packaging system, version A

A.1.1 Proxy-Based: Version A

The first system we analysed spread each algorithm and set of resources into individual packages that depends on MODELDEP, a proxy utility to access the resource files inside JAR files. If an algorithm depends on some resource file then it accepts as input parameter the model name (as a string) and uses the proxy to load the file. In this scenario pipeline packages have dependencies toward their engines but also toward specific models as shown in figure 4.

This system has several positive aspects. First of all, tests can be easily written by specifying potentially multiple resource packages as dependencies with Maven's test scope. Then, MODELDEP also defines a clear Java interface to define the contract for models and their versions. Additionally, the proxy system centralises the utility to load resources and therefore eases maintenance by not involving code duplication in several packages. However, models could not be swapped on the fly since it would involve recompiling and repackaging USING_MY_ALGO. Alternatively, to prevent modifications, USING_MY_ALGO could depend on both ADA and BOB packages but this would not be as flexible as we want it to be.

A.1.2 Proxy-Based: Version B

The second approach is based on Maven's version string: an ALGO will depend on an abstract MODEL which provides a mechanism to load a specific file from its JAR file and both ADA and BOB model versions are defined as subversion of MODEL. For example, the convention could be that if MODEL is defined as version 0.1 then ADA will use the string 0.1-ADA to define its version and the

ALGO will accept versions in the range $[0.1, 0.2)$.

The model actually used will be either specified at compile time (cf. USING.ALGO2) or be selected at runtime depending on the installed packages (cf. USING.ALGO1) as depicted on figure 5. While this offers a great flexibility to the user designing pipelines and allows him to swap models without repackaging anything, it means that test projects cannot ensure the correctness of more than one model version at a time. Moreover, if no specific version is bound to the pipeline, such as with USING.ALGO1, then there is no strong guarantee that a valid version is available at runtime.

A.1.3 Proxy-Based: Version C

The third version we studied is the most simple one: as illustrated on figure 6, instead of splitting everything in separate package, only engines are packaged independently of pipelines and resources, which are grouped together. On the one hand this system couldn't be simpler but on the other, since pipelines often involve several engines that are themselves used in several pipelines, the coupling of models and the corresponding algorithm implies that many packages have to be created to support each combination of models.

A.1.4 Proxy-Based: Version D

After exploring possibilities offered by the Maven packaging system we analysed how resources and engines are related to each others and used by pipelines in bluima. Figure 7 depicts those relations. It was also considered that repackaging is an acceptable cost to swap models. The most important point was avoiding at all cost to package an exponentially huge number of pipelines to match each and every possible combinations and for this reason version C was discarded.

We also reflected on the structure of the algorithm, especially on their input arguments. We came to the conclusion that, mostly for flexibility, they should accept InputStreams as input and not be bound to any models. Instead, the pipelines will be in charge to give them the proper resource data streams. Therefore the pipeline will have dependencies toward algorithms and models.

Finally, in order to centralise code and simplify loading resources we introduce ModelProxy, a utility class used by pipelines that, given a String representing a class name, opens a stream to a given file inside the class' JAR file.

In some cases, when processing the resource as a stream, it can be convenient to have access to its original filename (e.g. when reading a compressed archive). Therefore we encapsulate the name and the stream in a class – ModelStream – that can be seen as a specialised InputStream with a filename property.

A.2 Restructuring bluima

Now that the packaging strategy for engines and their resources has been designed we can actually apply it. It was decided to start with a relatively easy

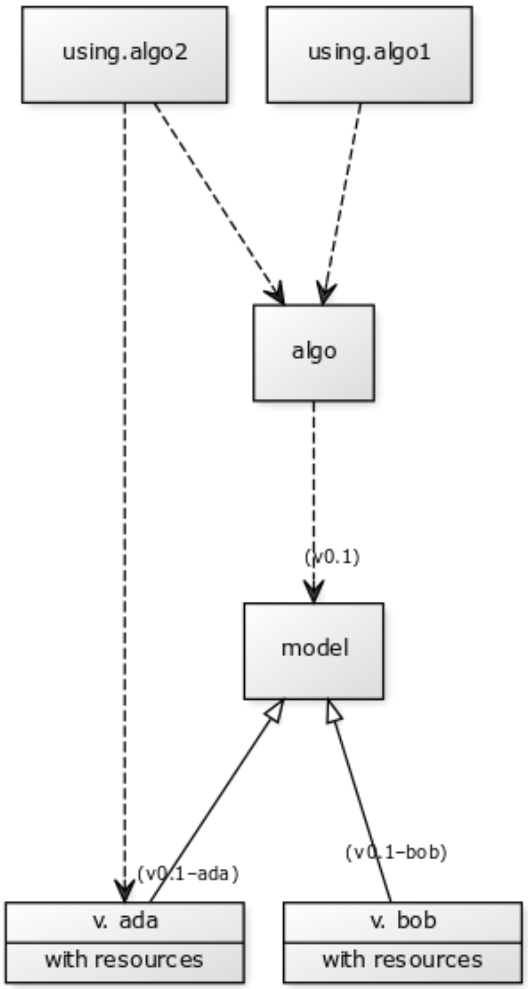

Figure 5: Proxy-based packaging system, version B

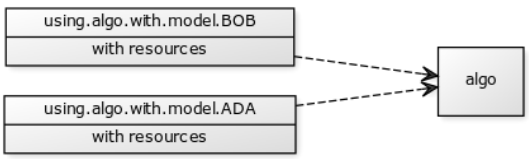

Figure 6: Proxy-based packaging system, version C

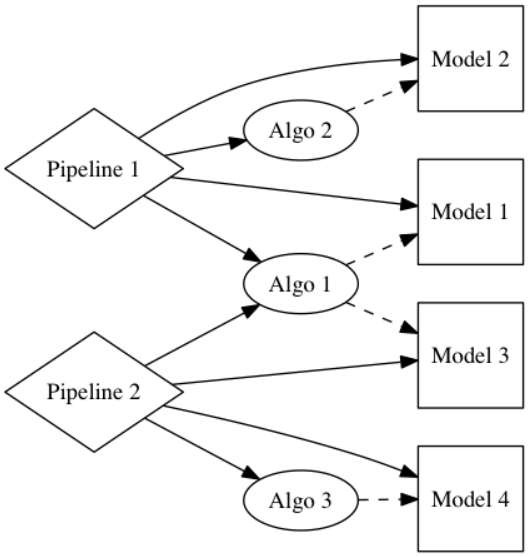

Figure 7: Proxy-based packaging system, version D

engine – Namely SentenceAnnotator which is part of OpenNLP – to make sure our previous decision was indeed viable. The approach was based on an iterative process that can be applied to other modules as well. But before diving into the details, we will first discuss how the different modules in bluima are related to each other.

We wanted to create a structured, yet flexible, file hierarchy for the engines and their resources. Since we are using Maven, we based our design on parent-child relationship in the pom.xml files: the root pom file, in addition to actually loading its children, defines the main settings, such as the Java version to use or the list of general dependencies, that will be shared with every of its children. The root pom file references both modules `modules/pom.xml` and `resources/pom.xml` that are responsible for loading their own children, that is the different engines or utilities for the former and the actual resource for the latter.

It follows that when running the install (or test) command on the root Maven project then every algorithms and resources are installed (or tested).

A.2.1 Case Study: Converting OpenNLP

When converting an engine such as SentenceAnnotator, the first thing to do is to identify the unit test responsible for its quality. In this case, the engine was converted from a classic UIMA [6] structure, based on XML description of engines and their configuration, to the more modern structure defined by uimaFIT [7], which allows a more compact pipeline runner.

Once the unit tests are properly working, we move on to the generation of the

resource packages. Since this phase is completely repetitive, we devised a script to make the developer's live less cumbersome: given a resource file, a resource name and a few other technical details we can produce a resource package, as described previously, that is ready to be used with an engine.

However, many engines are currently using Java FILE parameters, or Java STRINGs to identify files on the local filesystem. Therefore, we have to refactor these engines and their dependencies in order to use MODELSTREAM instead. The following strategy was applied to convert PoS, Token and Chunk engine of OpenNLP:

1. Renaming the PARAM_MODEL_FILE parameter (or any similar properties) of the engine into PARAM_MODEL.
2. Updating its INITIALIZE method to load a MODELSTREAM through MODELPROXY and the PARAM_MODEL parameter.
3. Updating the engine's dependencies such that they can be constructed from MODELSTREAM. Usually this plays well with the current implementation that must at some point use a FILEINPUTSTREAM. Hence, we can plug in our custom stream in place of the regular FILEINPUTSTREAM without restructuring the dependencies too much.
4. Updating the dependency list in the engine's POM.XML file to reflect any additional resource dependency.

Finally, at this last point the unit test should be updated to use resource package with PARAM_MODEL instead of absolute path to the resource file.

However, as discussed in section 4, the third step is not always trivial or quick to be implemented. The strategy defined in this section works well for relatively simple engines, such as the one from OpenNLP, or for engines with a well-modularised implementation. But for closed-source, more complex or engines with a defective software design that cannot withstand refactoring easily, this task becomes much harder, if not close to impossible to implement.

A.3 Configuration in Sherlok

Listing 7 illustrates how this strategy can then be used in Sherlok bundles configuration. Compared to listing 4 from section 4.4, here we have one extra Maven dependency (BLUIMA_SENTENCE_DETECTOR_GENIA) which contains the essentially two things: the model file used by SENTENCEANNOTATOR and a proxy utility class, GENIARESOURCE, that will be used to access the resource file.

```
{
  "name": "bluima.sentence",
  "domain": "examples",
  "version": "0.1",
  "dependencies": [
    {
      "value": "ch.epfl.bbp.nlp:bluima_opennlp:1.0.1",
      "value":
        "ch.epfl.bbp.nlp:bluima_sentence_detector_genia:0.1"
    }
  ],
  "engines": [
    {
      "name": "SentenceAnnotator",
      "class": "ch.epfl.bbp.uima.ae.SentenceAnnotator",
      "parameters": {
        "model": [ "ch.epfl.bbp.nlp.GeniaResource" ]
      }
    }
  ]
}
```

Listing 7: Configuration of SentenceAnnotator in a Sherlok bundle with the proxy-based packaging strategy

References

[1] Blue Brain Project, bluebrain.epfl.ch

[2] Sherlok, sherlok.io

[3] Bluima, github.com/BlueBrain/bluima

[4] Bluima: a UIMA-based NLP Toolkit for Neuroscience, by R. Richardet, Proceedings of the 3rd Workshop on Unstructured Information Management Architecture, Darmstadt, Germany, 2013, pp. 34–41, Gesellschaft für Sprachtechnologie und Computerlinguistik

[5] JSON format, json.org

[6] Apache UIMA, uima.apache.org

[7] Apache uimaFIT, uima.apache.org/uimafit.html

[8] Apache UIMA Addons and Sandbox uima.apache.org/sandbox.html

[9] Building Test Suites for UIMA Components, by P. Ogren and S. Bethard, Proceedings of the Workshop on Software Engineering, Testing, and Quality Assurance for Natural Language Processing (SETQA-NLP 2009)

[10] Design and implementation of the UIMA Common Analysis System, by T. Götz and O. Suhre, IBM SYSTEMS JOURNAL, VOL 43, NO 3, 2004

[11] Apache RUTA, uima.apache.org/ruta.html

[12] UIMA Ruta: Rapid development of rule-based information extraction applications, by P. KLUEGL, M. TOEPFER, P.-D. BECK, G. FETTE and F. PUPPE, Natural Language Engineering

[13] Apache Maven, maven.apache.org

[14] DKPro, code.google.com/p/dkpro-core-asl/

[15] A broad-coverage collection of portable NLP components for building shareable analysis pipelines, by R. Eckart de Castilho and I. Gurevych, Proceedings of the Workshop on Open Infrastructures and Analysis Frameworks for HLT (OIAF4HLT) at COLING 2014

[16] The Apache Software Foundation, www.apache.org

[17] Apache OpenNLP, opennlp.apache.org

[18] Linnaeus, linnaeus.sourceforge.net

[19] LINNAEUS: A species name identification system for biomedical literature, by M. Gerner, G. Nenadic and C. M. Bergman, BMC Bioinformatics 2010, 11:85

[20] BANNER: An executable survey of advances in biomedical named entity recognition, by R. Leaman and G. Gonzalez, Pacific Symposium on Biocomputing 13:652-663(2008)

[21] Dragon Toolkit: Incorporating Auto-learned Semantic Knowledge into Large-Scale Text Retrieval and Mining, by X. Zhou, X. Zhang and X. Hu, Proceedings of the 19th IEEE International Conference on Tools with Artificial Intelligence (ICTAI), October 29-31, 2007, Patras, Greece

[22] BIOADI: a machine learning approach to identifying abbreviations and definitions in biological literature, by C.-J. Kuo, M. Ling, K.-T. Lin, C.-N. Hsu, BMC Bioinformatics 2009, 10(Suppl 15):S7

[23] Language Detection Library for Java, by N. Shuyo, code.google.com/p/language-detection

[24] Java Libraries for Accessing the Princeton Wordnet: Comparison and Evaluation, by M. A. Finlayson, Proceedings of the 7th Global Wordnet Conference. Tartu, Estonia

[25] Exploiting Shallow Linguistic Information for Relation Extraction from Biomedical Literature, by C. Giuliano and A. Lavelli, Proceedings of the 11th Conference of the European Chapter of the Association for Computational Linguistics (EACL 2006), Trento, Italy, 3-7 April 2006

[26] JULES, UIMA Analysis Engine, julielab.de

[27] OSCAR4: a flexible architecture for chemical text-mining, by D. M. Jessop, S. E. Adams, E. L. Willighagen, L. Hawizy and P. Murray-Rust, Journal of Cheminformatics 2011, 3:41

[28] GENIA corpus, www.nactem.ac.uk/aNT/genia.html