Project Number: **770299**

NewsEye:

A Digital Investigator for Historical Newspapers

Research and Innovation Action
Call H2020-SC-CULT-COOP-2016-2017

D5.8: Personal Research Assistant: Explainer (b) (final)

Due date of deliverable: M45 (31 January 2022)

Actual submission date: 10 December 2021

Start date of project: 1 May 2018 **Duration:** 45 months

Partner organization name in charge of deliverable: UH-CS

Project co-funded by the European Commission within Horizon 2020		
Dissemination Level		
PU	Public	PU
PP	Restricted to other programme participants (including the Commission Services)	-
RE	Restricted to a group specified by the Consortium (including the Commission Services)	-
CO	Confidential, only for members of the Consortium (including the Commission Services)	-

Revision History

Document administrative information		
Project acronym:	NewsEye	
Project number:	770299	
Deliverable number:	D5.8	
Deliverable full title:	Personal Research Assistant: Explainer (b) (final)	
Deliverable short title:	Personal Research Assistant: Explainer (final)	
Document identifier:	NewsEye-T53-D58-Explainer-b-Submitted-v3.1	
Lead partner short name:	UH-CS	
Report version:	V3.1	
Report preparation date:	10.12.2021	
Dissemination level:	PU	
Nature:	Report	
Lead author:	Leo Leppänen (UH-CS)	
Co-authors:	Hannu Toivonen (UH-CS)	
Internal reviewers:	Axel Jean-Caurant (ULR), Johannes Michael (UROS)	
Status:		Draft
		Final
	x	Submitted

The NewsEye Consortium partner responsible for this deliverable has addressed all comments received, making changes as necessary. Changes to this document are detailed in the change log table below.

Change Log

Date	Version	Editor	Summary of changes made
28/03/2021	0.1	L. Leppänen (UH-CS)	First draft
05/04/2021	1.0	L. Leppänen (UH-CS)	Ready for internal review
06/04/2021	1.1	H. Toivonen (UH-CS)	WP leader's check
09/04/2021	1.2	J. Michael (UROS)	Internal review
14/04/2021	1.3	A. Jean-Caurant (ULR)	Internal review
16/04/2021	1.4	L. Leppänen (UH-CS)	Addressed review comments
16/04/2021	2.0	L. Leppänen (UH-CS)	Ready for quality management
28/04/2021	2.1	L. Leppänen (UH-CS)	Modifications based on quality management
29/04/2021	2.2	L. Leppänen (UH-CS)	Ready for submission
30/04/2021	3.0	A. Doucet (ULR)	Minor changes and finalisation
10/12/2021	3.1	A. Doucet (ULR)	Submission

Executive summary

This document describes the Explainer component of the NewsEye Personal Research Assistant. The Explainer provides a method for describing the historical newspaper analyses conducted by the Investigator component of the Personal Research Assistant in natural language, such as English, French, German or Finnish. By doing so, it facilitates a better understanding of how the results were obtained, how reliable they are and how the user might replicate the process. The Explainer is a natural language generation application. Its inputs are both a description of the computational steps conducted by the Investigator, as well as descriptors defining what heuristic led to the computation steps being taken. This input is fed into a pipeline of components that transforms the input into natural language. The individual parts of this pipeline are designed so as to be easily augmented and extended without needing to modify the surrounding parts of the pipeline. Likewise, the pipeline architecture is designed to allow relatively easy translation of the process to other languages by virtue of separating the domain-specific components from the language-specific components where possible. On both the abstract and implementation levels, the Explainer is strongly related to the Reporter component (see public Deliverable D5.7) and shares a large portion of its code base with it.

Contents

1. Introduction

Historical newspapers collect information about cultural, political and social events in a more detailed way than any other public record. At the same time, analysing the wealth of information in the newspaper archives has traditionally been difficult and time-consuming. The NewsEye project develops methods and tools for effective exploration and exploitation of historical newspaper archives.

The core concept of NewsEye is a set of tools and methods, from text recognition to automated exploration of texts, that improve the users' capability to access, analyze and use the content of historical newspapers, stored in digital libraries (Figure 1).

This document describes some functions of the Personal Research Assistant developed as part of the NewsEye project. The Assistant carries out automated, iterative analysis of corpus content and reports on the results, functioning as the user's intelligent and transparent aid.

In this deliverable, we will first describe the Personal Research Assistant and the Explainer's role in it. We will then present a brief overview of the state of the art in the field of natural language generation in Section 3, highlighting how the methods employed can be divided into few archetypical approaches. Section 4 then presents an analysis of the interplay of the system requirements and the state of the art, identifying a suitable overall architecture for the system. The bulk of the deliverable, Section 5 presents the implementation of the Explainer. Sections 6 and 7 describe how the Explainer integrates with the rest of the personal research assistant as a service and on a more granular level. Finally, Sections 8 and 9, respectively, presents an evaluation of the Explainer and some concluding thoughts.

This document is a complete description of the Explainer, meaning that it also contains information about the aspects of the system that are shared in totality with the Reporter (see Deliverable D5.7) and with the previous iteration of the Explainer described in the non-public deliverable D5.5. Where relevant, segments of text have been shared across the deliverables. Readers who are already familiar with the Deliverable D5.7 describing the NewsEye Reporter will find the contents of Sections 1 through 3 familiar, and will identify that the requirement analysis conducted in Section 4 reaches conclusions highly similar to those presented in Deliverable D5.7 describing the Reporter component. In terms of concrete changes to the system since the non-public Deliverable D5.5, the Explainer has been extended to support all the analytical tools and decision making heuristics used by the Investigator. The Explainer has also been extended in terms of the selection of languages it can produce, and can at this point produce text in English, Finnish, German and French.

2. The NewsEye Personal Research Assistant

This deliverable describes the Explainer, a part of the larger system known as the NewsEye Personal Research Assistant ('Assistant'). In addition to the Explainer, the Assistant consists of an Investigator component, a Reporter component and an additional Controller component. We next give an overview of these components, followed by a more in-depth description of the communication between the Investigator and Explainer components.

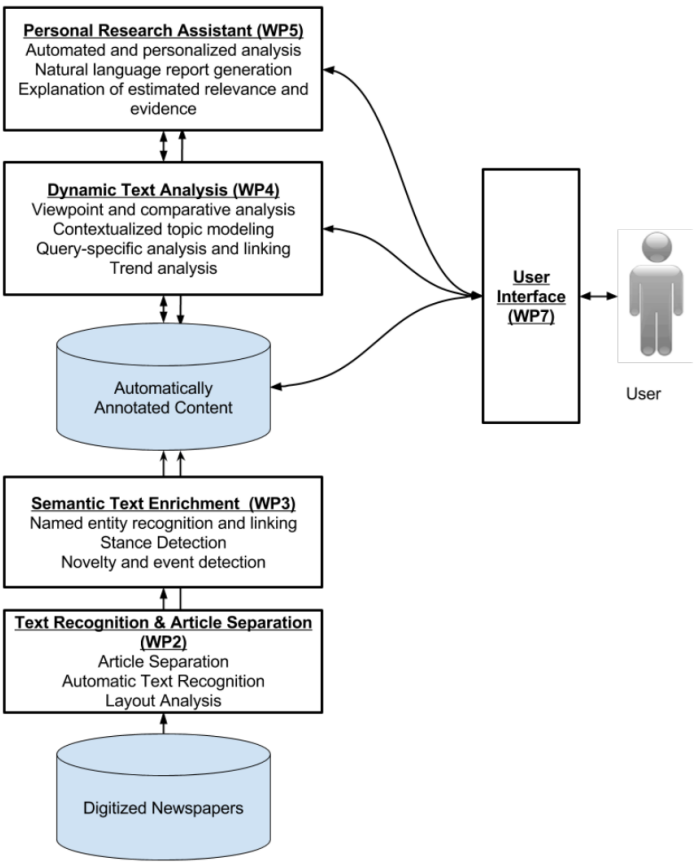

Figure 1: An overview of the NewsEye concept. This document describes some functions of the Personal Research Assistant (Work Package 5).

2.1. An Overview of the Personal Research Assistant

As noted above, the personal research assistant consists of three primary components (Investigator, Reporter and Explainer) and a Controller component. The Controller component has two primary functions. First, it provides an Application Programming Interface (API) for users and user interfaces (UIs), especially the NewsEye Demonstrator (see Deliverable D7.8). This allows outside users to view the Assistant as a single, unified, system so that they do not need to concern themselves with the internal division of labor within the Assistant. The API is used via HTTP(S) queries and is described in more detail in Deliverable D5.6, which describes the present version of the Investigator. Second, as the name implies, the Controller provides a central control mechanism that passes messages and results between the three major subsystems of the Assistant, i.e. the Investigator, the Reporter and the Explainer.

This design facilitates the distribution of the Assistant components over multiple virtual or physical machines if such a distribution would become needed due to increasing amounts of users. Modifying the Assistant so that a single Controller instances acts as a load balancer and distributor of work to multiple instances of the subcomponents, while not trivial, would be possible following the industry standard approaches used in many other online services.

The Investigator component, in broad terms, autonomously performs a series of queries over a newspaper corpus using different tools provided by Work Packages 3 and 4 to identify potentially interesting factors from the data. The present version of the Investigator is detailed in Deliverable D5.6. The analytical results obtained by the Investigator are passed via the Controller to the Reporter component described in Deliverable D5.7.

Having received the analytical results from the Assistant Controller, the Reporter then transforms the results into a natural language document describing the most salient factors of the results. The resulting natural language document is then returned to the Controller. The Controller then sends the document to the party that requested it. In the most likely scenario, this is the NewsEye Demonstrator that displays it to the end user.

In addition to this, the Assistant contains a database component which stores analysis results obtained from the Investigator, the reports obtained from the Reporter, explanations obtained from the Explainer and other necessary data. Section 2.2 describes the way the Investigator and the Explainer are connected in order to produce and store the explanations produced by the Explainer.

Finally, the Assistant also contains the Explainer component, described in the present deliverable. Whereas the Reporter describes *what* the Investigator found in the corpus, the Explainer described *how* those findings were obtained and *why* the Investigator conducted the analyses that resulted in them being obtained. In other words, whereas the Reporter describes the end result of a process, the Explainer describes the process itself.

An overview of the Personal Research Assistant's architecture is presented in Figure 2.

2.2. The interaction between the Investigator and the Explainer

The interaction between the Explainer and the Investigator is driven by the Controller. After an analysis task is complete, or when an explanation is specifically requested, the Controller will send a request to the Explainer component for producing a natural language explanation of the computation process con-

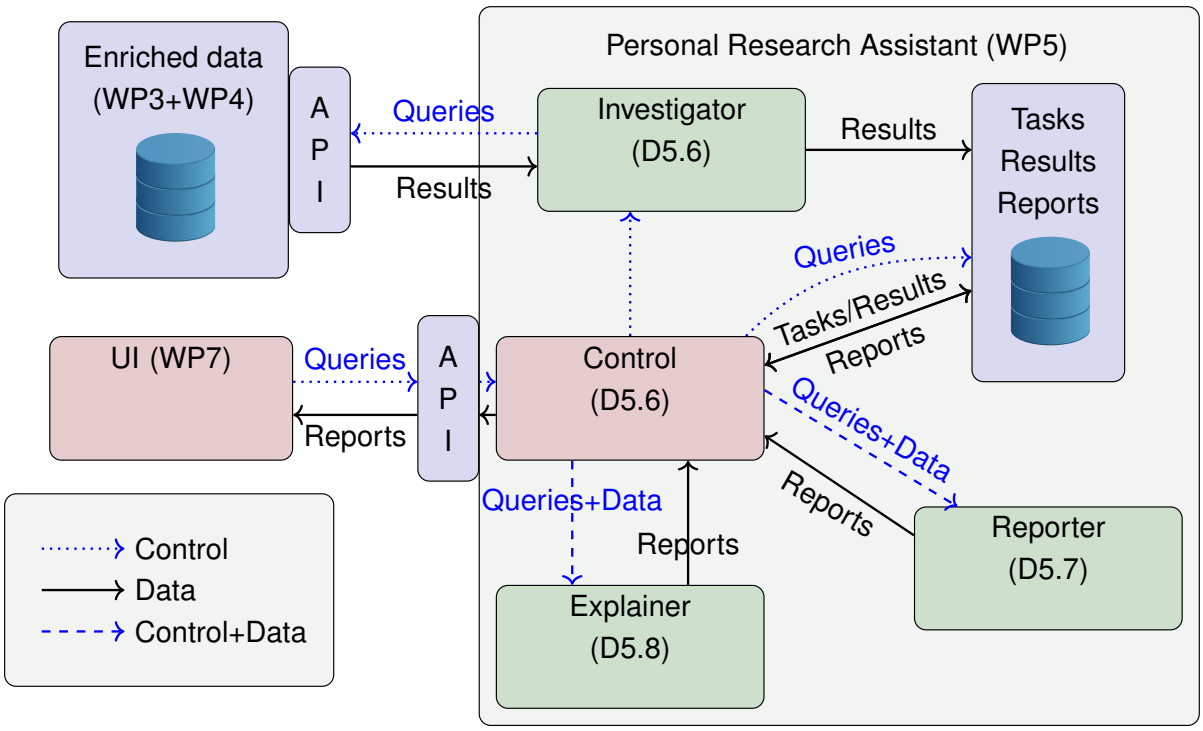

Figure 2: Flow of requests and data between the components of the Personal Research Assistant and the associated components developed in Work Packages 3, 4 and 7.

ducted by the Investigator in obtaining some specific result. The Explainer then produces such a natural language explanation and returns it to the Controller, which adds it to the Assistant's database, from where it can be retrieved and returned to the user. This flow of information and requests is described in Figure 2.

Notably, the Explainer does not report on the results that were found, but merely on the process of finding them. Similarly, it does not – at least in the present – attempt to present any type of a simplified view of the Investigator's actions: the goal is to allow the user to replicate the steps taken by the Investigator, and thus the explanation should not skip any steps. Furthermore, the Explainer does not conduct any post-hoc explanation about *why* the steps were taken: this reasoning comes from the Investigator itself and the Explainer merely describes it in natural language.

3. Natural Language Generation

The general task conducted by the Explainer is known as natural language generation, or NLG for short. More specifically, the Explainer is performing 'data-to-text NLG', where 'data' refers to structured data. That is, the Explainer is not designed to ingest unstructured data, such as raw text. In this section we first give an overview of (data-to-text) NLG in general as well as how it can be viewed as a series of subtasks, followed by a description of how the subtasks common to data-to-text NLG can be completed.

3.1. Natural Language Generation as a Process

A large number of data-to-text systems have been reported in the literature, in different domains, with varying types of input data [1]. For example, BabyTalk [2] is an NLG system that generates medical

reports from sensors monitoring babies in Neonatal Intensive Care Units. Hallett and Scott [3] describe a system for generating reports from events in medical records. Several systems have been developed that generate weather forecasts from the output of weather computer simulation models [4, 5, 6]. Other generate summaries from employment statistics [7, 8]. In addition to these efforts, several commercial NLG systems exist in a number of domains. Among the larger commercial players that provide NLG products and/or services are Automated Insights, Arria, AX Semantics, Narrativa, Narrative Science, and Yseop [9]. As demonstrated by the amount of commercial entities working in the domain, NLG technology is of increasing interest even without the academia, for example in the newsroom, even if it is not always clear to the non-technical stakeholders how to best employ the technology [10].

General data-to-text NLG systems normally involve three processes: deciding what to say (content determination), how to organize it (document and sentence planning), and how to express it (surface realization) [11].

In general, the content determination process of a data-to-text system would entail subprocesses such as computing and deriving new information from system input. For example, an input consisting of a table of some daily measurements might be augmented with monthly averages, changes over time spans, etc. In the specific case of the Explainer, however, this stage is not meaningfully present, as all the input data is provided by the Investigator and no data augmentation is conducted.

Document planning is a stage that produces 'messages' – pieces of information that are meaningful in isolation and could be conveyed to the reader via the final text – and organizes them into a structure that defines in which order they should appear in the final document. This initial version of the *document plan* is based solely on the information of the messages and can be modified in the process for better linguistic fluency. For example, if the document planning phase decides to place two pieces of information sequentially in the document plan, a further stage of the generation process might place a third piece of information between them if it makes the resulting text more fluent.

The following stage, microplanning, organizes the lower-level structures of the document. This includes, for example, deciding what linguistic expressions are to be used to express each message, how domain entities are referred to, whether multiple messages can be expressed in a single sentence, etc.

Finally, the realization stage takes as input the document plan and produces the final text output. This includes inflecting the words selected in the microplanning stage to the correct, grammatical forms and ensuring the text is orthographically correct, and adding any markup needed to display the text in the target medium.

Finally, we emphasize that the above characterization is more of an aide for reasoning about the types of decisions NLG systems have to complete. Previous NLG systems have employed a wide variety of techniques that can make the distinctions between the aforementioned phases of the generation process fuzzy or even remove them altogether [12].

3.2. Methods for Data-to-Text Natural Language Generation

In addition to different ways of dividing the larger generation tasks to smaller subtasks, there are also several competing methods for completing said subtasks [12]. In broad terms, we identify two main approaches: rule-based approaches and trainable approaches.

Rule-based systems are based on handcrafted rules, corpus analysis and expert consultations [13]. They are usually more robust than trainable approaches and are widely used in industry and for commercial purposes. As the rule-based approach is finely controlled, the output can be guaranteed to be more understandable by humans. Rules also provide relative high guarantees of correctness, and in case of errors, can be easily corrected by modifying the system's source code. At the same time, rules are limited, especially when the domain complexity increases and the generation of the rules can be an expensive effort requiring significant input from domain experts.

Trainable approaches, such as neural networks, reinforcement learning, or Hidden Markov Models are more flexible, easier to develop, and more domain-independent. However, one of the challenges of these trainable approaches is the lack of sufficient quantity of aligned datasets that can be used to derive rules or train the NLG system [13]. Even when the data is available, the expected output text is often not aligned with the input data, and thus cannot be used directly for the development of an NLG system [14].

At the onset of the NewsEye research process, we identified that the state of the art in these trainable approaches also seemed to suffer from a series of further problems that were relevant for the NewsEye context. First, purely trainable approaches struggled to reach the linguistic depth of their competitors [15], with even the then-most-recent trainable end-to-end architectures failing to conclusively outperform rule-based approaches even in a relatively simple and constrained generation task when evaluated by humans [16]. Second, most trainable approaches suffered from a lack of transparent generation process. This chiefly manifested in the difficulty of detecting and correcting system errors: it was exceedingly difficult to guarantee any specific level of correctness for an NLG system based on neural networks, and in case of problems in the system it was not possible to conduct surgical modifications. Rather, especially in case of (deep) neural networks, the only solution was to further train the system with more training data. Whether this retraining and fine-tuning results in some unknown pathological behaviour in some corner cases would be difficult if not impossible to determine. Third, empirical evidence indicated that such systems were – and continue to be – prone to overfitting to the training data, which manifests as '*hallucination*', where the system produces output that is not grounded in the underlying data [17]. These problems were made more complicated by the fact that automated evaluation of an NLG system's quality was – and continues to be – an unsolved problem, with evidence suggesting that the most popular automated metrics fail to properly correlate with the judgments of human evaluators [12, 16, 18, 19].

Our interpretation of the current state of the art in NLG at the onset of the NewsEye research project was thus that trainable approaches were mainly ready for real-world use in situations where either the produced texts were very short (i.e. scenarios similar to the E2E Challenge described by Dušek, Novikova, and Rieser [16]) or in scenarios where even major mistakes in individual pieces of output are not problematic. While many of the problems identified above have seen significant scholarly attention, with attention being directed especially towards the hallucination problem [20, 21], the present state of the art in NLG has not significantly modified the above analysis. For example, hallucination continues to plague even state-of-the-art neural systems [22, 23]. As such, we do not believe the present state of the art in NLG is sufficiently different from that at the onset of the NewsEye project, and thus our requirement analysis (see below) conducted at the onset of the project remains valid.

4. Requirement Analysis

The NewsEye Personal Research Assistant is intended to be used for exploration of historical newspaper corpora. Part of the intended user base is formed by academics such as historians and social scientists. The role of the Explainer in the Personal Research Assistant is especially crucial. The Explainer is intended to enable the users to understand the analytical processes conducted by the Investigator and reported by the Reporter, enabling them to verify that the results are meaningful and correct. Furthermore, it is also intended to enable these users to replicate the processes by themselves to ensure the results' correctness and to derive further results. As such, it is absolutely crucial that the Explainer's output is *correct*. Consequently, this requirement speaks for a rule-based approach.

As a consequence of the role of the Explainer in the larger Assistant, it does *not* necessitate high linguistic *variation*. The Explainer is a tool for understanding how other texts were produced, and is therefore used to obtain short and correct information: the users are not expecting high prose but concrete details in a simple and understandable format. This requirement speaks towards a rule-based approach, as the main benefit of data-driven methods is not needed.

The NewsEye Personal Research Assistant is also intended to be easily *extensible* so that it can take advantage of new, improved, analytical tools as they become available. Adding such a new tool should be straightforward and should not invalidate previous work. As the Explainer needs to adapt to each such additional tool, this requirement speaks towards a modular system, where each analytical tool is accompanied by a small module that can be incorporated into the Explainer, handling the specifics of said tool.

The Explainer should, also, be able to produce the explanations in *multiple languages*. While this requirement does not specifically *require* any specific approach, taken together with the *extensibility* requirement it speaks for a modular approach. A modular system can be constructed so that the language-independent parts of the generation process can be shared by the different languages. This further improves the extensibility of the system by allowing the system to be extended to new languages without having to duplicate all of its components.

As a summary, we identify the following requirements and their implications for the system:

- Correctness - Suggests a rule-based approach
- Low variation - Suggests a rule-based approach
- Extensibility - Suggests a modular approach
- Multilinguality - Suggests a modular approach

As a whole, the requirement analysis suggests that the Explainer should be a modular system based on human-produced rules. This analysis is further supported by the lack of any suitable training data set that would be required for a trainable approach to be feasible.

At the same time, the requirement analysis does not completely forbid the use of trainable methods as *parts* of the larger system in settings where they are unable to significantly affect the correctness of the output. Early results in other domains indicate that dividing the unified, end-to-end, neural NLG model into separate, but still neural, subcomponents increases the performance of the system [24]. While the limitations of such models are far from known, even these early successes indicate that a hybrid system

employing both rule-based and neural modules could be successful. The architecture of the Explainer facilitates the inclusion of neural components if they are developed.

5. The Explainer Architecture

Based on the above requirement analysis, we decided to base the Explainer on the same architecture as the Reporter (see Deliverable D5.7), which in turn is based on the multilingual news report generator previously produced by the University of Helsinki Department of Computer Science [25]. The modified architecture (see Figure 3) is formulated as a pipeline where raw data and relevant parameters are fed in at the start. This input is then processed through a pipeline of individual components, where the components of the pipeline each modify the input towards the final output. At the end of the pipeline, a finished natural language text document is produced as the output. The current version of the Explainer produces text in English.

In the next sections, we will step through the generation process, discussing each pipeline component in turn.

5.1. Input: Events, Facts and Messages

Each run of the Explainer produces a textual description of how some specific analytical result was arrived at by the Investigator. Viewing the investigation process as a computational graph such as that shown in Figure 4, each run of the Explainer produces an explanation detailing *how* and *why* a specific result (the non-circular nodes marked with 'R' in Figure 4) was arrived at from the initial user input (the root of Figure 4).

For example, to explain the bolded result marked R' in Figure 4, the Explainer needs to express, at minimum, the bolded edges that define the process of how the result was derived from the initial user input. This bolded path of edges is the *critical computational path* of the specified result. It is defined in terms of a sequence of discrete Events, which correspond to the edges of the graph. The Events describe both what analytical task was compeleted by the Investigator, and the Investigator's reason for undertaking said task.

This sequence of Events is the primary input of the Explainer, and is provided to the Explainer in the JavaScript Object Notation (JSON) format via a HTTP(S) request. The JSON format is an industry standard format for moving data between services using HTTP(S). Figure 5 shows an excerpt of what the system input looks like.

As with the Reporter (see Deliverable 5.7), the Explainer refers to information internally in two primary formats: as *facts* (see Section 5.1.1) and as *messages* (see Section 5.1.2). A fact represents a distinct point of information. Each Event produces two facts, one describing what task was completed by the Investigator and the other the reason for undertaking that task. Facts are the minimal pieces of information that are expressible to a human reader and the smallest informational units used by the system. Each fact is wrapped into a 'message', which provides a place for any mutable data that needs to be associated with a specific fact for the generation process to succeed.

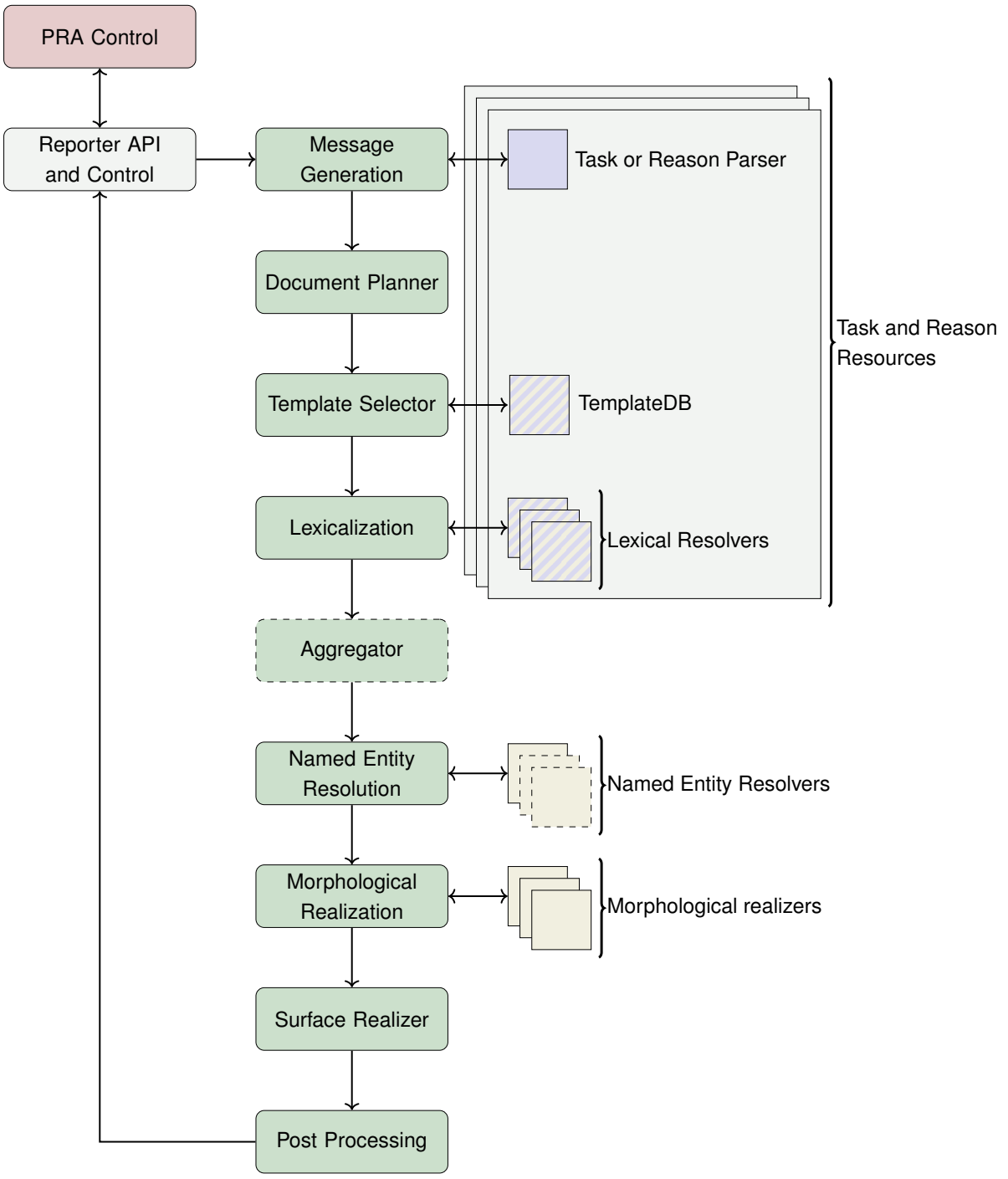

Figure 3: High-level architecture of the Explainer. Sharp-cornered elements represent various resources available to the Explainer. Sand-colored elements (e.g. Morphological realizers) are language-specific but not directly related to any specific analytical tool. Sapphire-colored elements (e.g. Task Parsers) are specific to an analytical tool or a heuristic used by the Investigator, but not to any language. Dashed elements (e.g. TemplateDB) are specific to both. The green middle column forms the main NLG pipeline. Dashed borders indicate locations of components that are not meaningfully present in the current iteration of the Explainer, but that could be introduced easily using similar components used e.g. in the Reporter.

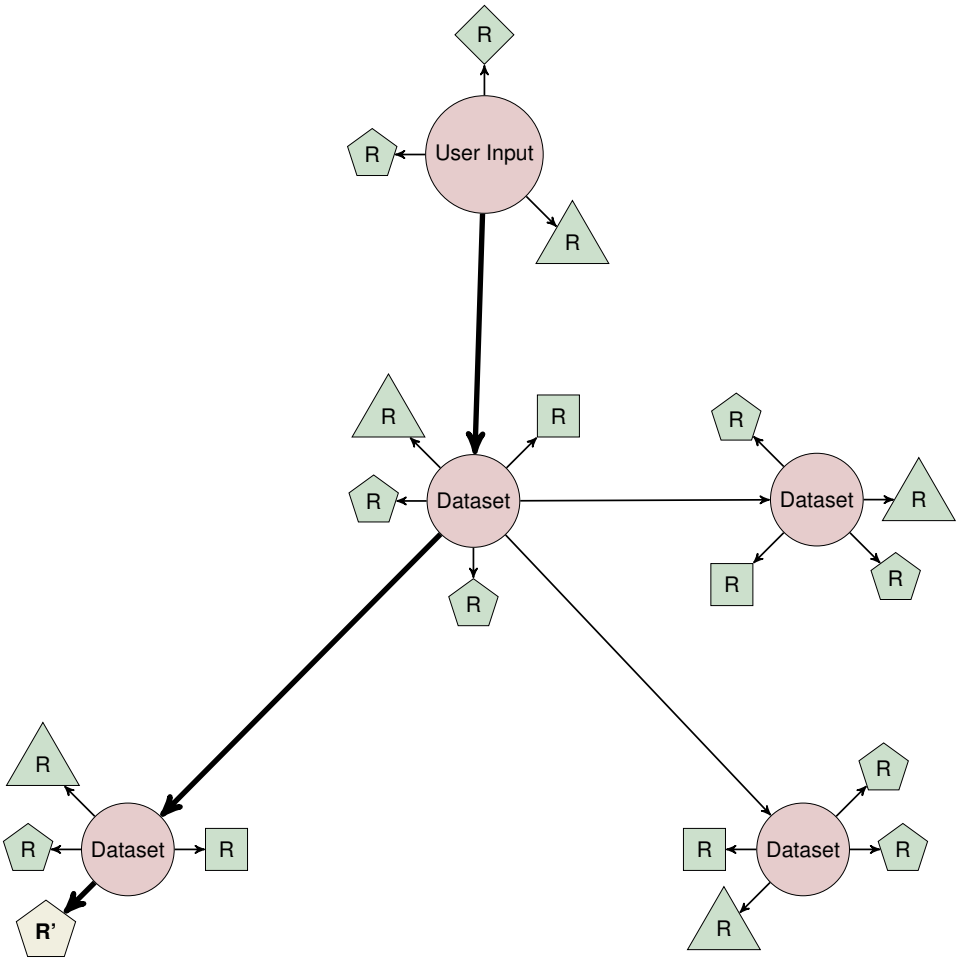

Figure 4: The Investigation process as a computational tree. Each circular node represents a specific dataset of newspaper articles, with the adjacent nodes of various shapes (denoted by 'R' for 'result', with the different shapes indicating the results are not of uniform type) representing a result of conducting an analysis on said dataset. The edges between the corpora indicate processes that derive one dataset from another and the edges between a corpus and a result designate the analytical process that leads to the result. The process starts from the top-most dataset, which is the initial input provided by the user. The bolded edges designate actions ('tasks') of the Investigator that lead to the procurement of the result R' from the initial dataset. For additional details on the process, see Deliverable D5.6, which uses a notation slightly different from that used above.

```
[
  {
    "id": 6019,
    "reason": {
      "name": "initialization"
    },
    "task": {
      "name": "ExtractFacets",
      "uuid": "227542cd-d57f-4af4-87a9-121beccef383"
    }
  },
  {
    "id": 6019,
    "reason": {
      "name": "initialization"
    },
    "task": {
      "parameters": {
        "unit": "tokens",
        "max_number": 30
      },
      "name": "ExtractWords",
      "uuid": "47402224-cac4-4482-8642-cf87e9f0f0c9"
    }
  },
  {
    "id": 6019,
    "reason": {
      "name": "initialization"
    },
    "task": {
      "parameters": {
        "unit": "tokens",
        "max_number": 30
      },
      "name": "ExtractBigrams",
      "uuid": "def01b23-9039-4111-acb8-26566bbb0c20"
    }
  },
  ...
]
```


Figure 5: An example of the input to the Explainer as provided by the Investigator. Note that the data structure has been edited for length and some details are omitted.

Field	Example value
type	task
name	ExtractBigrams
parameters	[ExtractBigrams:UNIT:tokens]
id	1

Table 1: The fields of the 'fact' data structure and an example of their possible values. This example fact would correspond to the idea that the distribution of the bigrams – of tokens rather than e.g. stems, as per the `parameters` field – were queried by the Investigator. The reasoning for this task would then be provided in a second message with a type `reason` and the same `id` value.

5.1.1. The Fact data structure

The fields of the Fact data structure are detailed in Table 1, together with example values. Whereas the Reporter required a fairly complex Fact data structure, the Explainer's Fact only contains three primary fields, `type`, `name` and `parameters`, as well as an `id` field.

The simplest of the fields is the `type` field, which encodes whether the fact represents a task or the reason for said task being completed.

In case of a *task* type message, the `name` field encodes which task was completed. For example, in the example in Table 1, the analysis being conducted is the `ExtractBigrams` task. In case of a *reason* type message, the name could be, for example, `Initialization` to indicate that the related task was completed as one of the tasks that is always taken when the system starts analyzing a corpus.

The field `parameters` encodes any per-task or per-reason parameters that need to be expressed for the explanation to make sense. For example in Table 1, the field encodes the information that the bigrams were extracted from a tokenized text, rather than a stemmed text.

Finally, the `id` field allows the Explainer to link the task and the reason of a single event, as both share the same id value. The ID value also encodes the temporal ordering of the events, as IDs are allocated in order.

During the processing performed by the Explainer, the fact itself is an immutable piece of information. In other words, the Explainer never modifies the fact itself once it has been created. This ensures that the underlying analysis results remain unchanged through the generation process.

5.1.2. The Message data structure

As noted above, the fact data structures are considered immutable and the generation is not to modify them in any way. In fact, they are technically implemented in a way that makes it impossible for them to be modified by accident. This ensures the underlying information within the generation always stays true. At the same time, the system needs to attach information to the facts, such as what *template* (see Section 5.3) is to be used to express the fact. For this reason, we encapsulate the immutable facts in mutable data structures called 'messages'. In the present version of the Explainer, the message data structures contain the fields shown in Table 2.

Field	Description
facts	The facts related by this message. By default contains only a single fact, could contain multiple e.g. as a result of aggregation.
template	The template associated with the message. Initially empty, gets assigned during template selection.

Table 2: The fields of the 'message' data structure and a description of their contents.

Beyond the mutability, another notable distinguishing feature between fact and messages is that a single message is allowed to contain multiple facts. This could happen as a result of, e.g., the aggregation phase where multiple templates are combined into a single template which then expresses multiple facts. As the present version of the Explainer does not conduct aggregation (we found it unnecessary given the task and the language used), this property is currently unused but retained for ease of code-reuse between the Explainer and the Reporter.

5.1.3. Fact and Message Generation

The facts and messages are generated from the input provided by the Investigator in the message generator. It ingests the JSON input and outputs the messages and facts, which are in turn ingested by the document planner.

Notably, the specifics of how an analysis result is to be parsed is dependent on the precise analytical tool being described. For example, the metadata contained in the `parameters` fields of the input are analysis-specific. This means, in practice, that the parsing functionality of the message generator needs to be adjusted every time the analytical tools used by the Investigator are altered and when a new analytical tool is integrated into the Investigator. Similarly, the process needs to be modified whenever a new heuristic is implemented in the Investigator. As such, the message generator is constructed so that it delegates the parsing to a set of individual parsers, each of which contains a method for identifying the relevant subset of tasks or reasons it can parse. These individual parsers are depicted in Figure 3 as the small box with the label 'Task or Reason Parser'.

Task parsers are defined on the level of the analyses conducted by the Investigator. For example, the Investigator might implement an analysis called '*GenerateTimeSeries*' which has its own unique set of parameters and as such needs its own, individual, logic. This logic is encoded in a single Task Parser specific to this analysis. The task parsers return messages, wherein the `type` of the message is '`task`'. Similarly, Reason parsers are defined for each heuristic the Investigator has, and parse the details of the `reason` fields of the underlying input. They produce messages wherein the `type` of the message is `reason`.

Processing the list of task-reason pairs provided as input in sequence (see Figure 5), the message generator identifies for each task-reason pair both a Task Parser and a Reason Parser. Using these, two messages are then generated from the pair, with distinct types. The two generated messages make up a tuple, all of which are collected and form the output of the procedure. This procedure is described below in pseudocode as function GENERATEMESSAGES in Algorithm 1.

Algorithm 1 Pseudocode describing the relation between the tasks conducted by the Investigator (see Figure 5) and the task parsers.

function GENERATEMESSAGES($Event$, $TaskParsers$, $ReasonParsers$)
  $Messages \leftarrow []$
  for all $Event \in Events$ **do**
    for all $TaskParser \in TaskParsers$ **do**
      if $TaskParser$ is applicable to $Event$ **then**
        $TaskMessage \leftarrow$ APPLY($Event, TaskParser$)
        break
      end if
    end for
    for all $ReasonParser \in ReasonParsers$ **do**
      if $ReasonParser$ is applicable to $Event$ **then**
        $ReasonMessage \leftarrow$ APPLY($Event, ReasonParser$)
        break
      end if
    end for
    $Messages.append((TaskMessage, ReasonMessage))$
  end for
  return $Messages$
end function

This formulation of the message generator allows for significant decoupling of the Explainer's core functionality from the specifics of individual tools and heuristics, thus enabling easier modifiability and simpler integration of any future tools.

This list of tuples produced by the message parsing is provided as input to the next step in the pipeline.

5.2. Document Structuring

The tuples of message data structures, which contain the fact data structures, are provided as input to the next component in the pipeline, the document planner. Whereas in the Reporter the document structuring is a highly complex task, in the Explainer it is very simple.

The Explainer's output needs to reflect the temporal ordering of the tasks taken by the Investigator: it is not meaningful to describe a task T_n before any of its preceding tasks T_{n-k} for any $k > 0$. Similarly, both a task and its reason need to be discussed together. As such, the document planner of the Explainer simply produces a document plan containing the task-reason pairs in the temporal order they were conducted in, as defined by the `id` fields. It produces a tree-structure where said messages are the leaves.

The majority of the complexity in the Reporter's version of this stage was caused by a need to group the messages into logical paragraphs consisting of multiple related messages. In the case of the Explainer, we observe that each task-reason pair on its own makes up sufficient content for a paragraph, and thus each paragraph consist of only a single such pair.

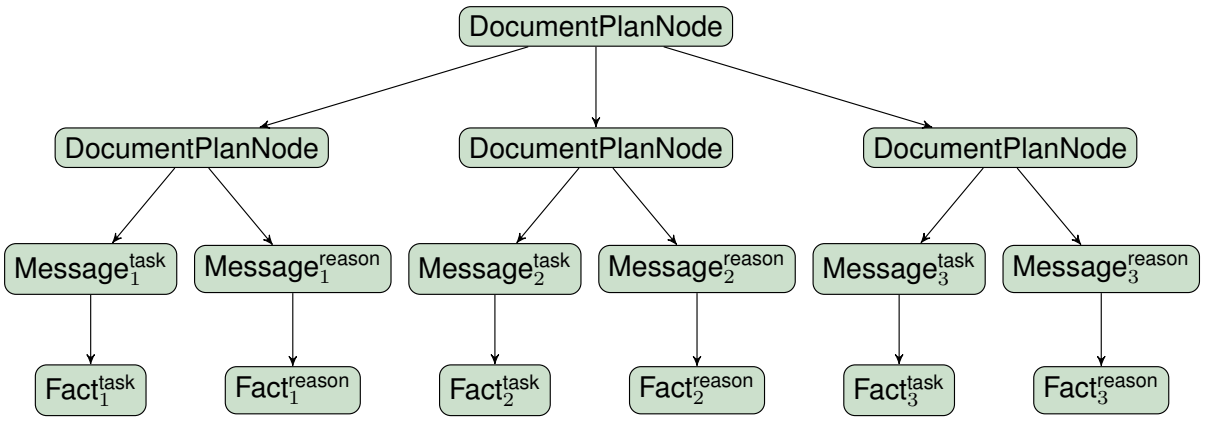

Figure 6: An abridged version of a Document Plan as produced by the Document Planner.

```
en: All pairs of subsequent {parameters} were extracted and counted.
fi: Kaikki {parameters} noudettiin ja laskettiin.
| name = ExtractBigram
```

Figure 7: An example of the templating language.

This process thus results in a document plan like that shown in Figure 6, which acts as the input for the next stage of processing.

5.3. Templates and Template Selection

As the next step in the NLG process, the language-independent messages need to be transformed to some type of linguistic constructs. In the case of the Explainer, these language constructs correspond to individual relatively long expressions, potentially multiple sentences long each, and are provided to the system as *templates*.

Broadly speaking, the amount of templates needed by the system for each language scales as $\mathcal{O}(V|H|+V|A|)$, where $|H|$ is the number of heuristics supported by the system, $|A|$ is the number of analytical tools supported by the system and V is a linguistic variation coefficient denoting in how many distinct ways the system should be able to express each heuristic and analysis. As linguistic variation is not a very important factor for the Explainer, V is expected to be very low, and initially strictly 1.

The templates are provided to the system in a custom templating language, of which Figure 7 provides an excerpt. As can be observed, a template group in our system consists of three parts:

1. A per-template language identifier such as '`en`' for English,
2. A phrase (template) expressed in natural language with slots (indicated by curly brackets `{}`) that can be filled with information from the Fact data structures, and
3. conditions for using the templates in the group, usually that it applies to a specific task.

Moreover, slots in the templates can be optionally associated with grammatical cases and other additional information to instruct components further 'down' the pipeline on how to treat them.

The templating language was designed to allow for multilinguality in the system. Multilinguality is supported by allowing expressions in different languages to be specified within the same template group, i.e. by adding 'fi', 'en' or 'de' at the beginning of the template as shown in Figure 7. In many cases, adding new languages to the template group does not require creation or modification of the conditions of the group. It only requires the translation of the template text, as demonstrated by Figure 7.

The aim of the templating language is to make it relatively simple for domain experts to contribute to the creation of templates without significant background in linguistics or natural language processing, as would be required if the templates were expressed as, for example, parse trees.

We note here that the templates are, fundamentally, related to the tasks and heuristics in the same way as the parsers are. As a consequence, the templates are provided together with the relevant parser in a joint resource unit, called either a Task Resource or a Reason Resource depending on whether the parser is a Task Parser or a Reason Parser. Also included in these resources are the lexical parsers, discussed in Section 5.4.

To select a suitable template for a given message in the document plan, the system first finds, for each fact, a template that can be used to express said fact. In the case that multiple valid templates exist for a certain fact, one is selected pseudo-randomly. While no such cases exist in the present version of the Explainer, this ability to define multiple valid templates allows for some linguistic variety in the output if desired. The randomness used in template-selection is pseudo-random in the sense that the random number generator is re-initialized with a known constant starting position (a 'seed number') for every generation task. This means that every time the system is called with the same set of inputs it produces the same 'random' choices. This is useful in that it produces variety into the text for the human reader of the resulting report, but still makes the process deterministic for most relevant purposes insofar as development is concerned. This also means that if the same generation task is run multiple times, the same report is produced down to the minor linguistic choices.

Having identified a suitable pseudo-random template, a copy of the identified template is then attached to the tree as a child of each fact's parent message so that each slot of the template contains a link to the underlying fact. Individual slots rather than the whole template are associated with the facts, since the following aggregation phase can (and usually does) result in sentences referring to multiple facts. This phase results in a tree-like structure such as the one shown in Figure 8.

Finally, the resulting document plan (with added templates) is provided to the lexicalization component for further processing.

5.4. Lexicalization

The templates attached and filled by the previous stage of the pipeline can still contain unlexicalized content in the slots of the templates. These unlexicalized segments most commonly come in the form of the various sub-elements of the `parameters` fields. To handle these, the system allows for inclusion of *lexical resolvers*.

These lexical resolvers, however, are not limited to just working with tags, but can be applied to transform any token into one or more tokens. The resolvers are applied iteratively until the sentences stabilize, meaning that a lexical resolver's output can be processed further by another lexical resolver. They thus give a significant increase in the expressive power of the system. This power, however, comes at

Figure 8: Part of a Document Plan template selection.

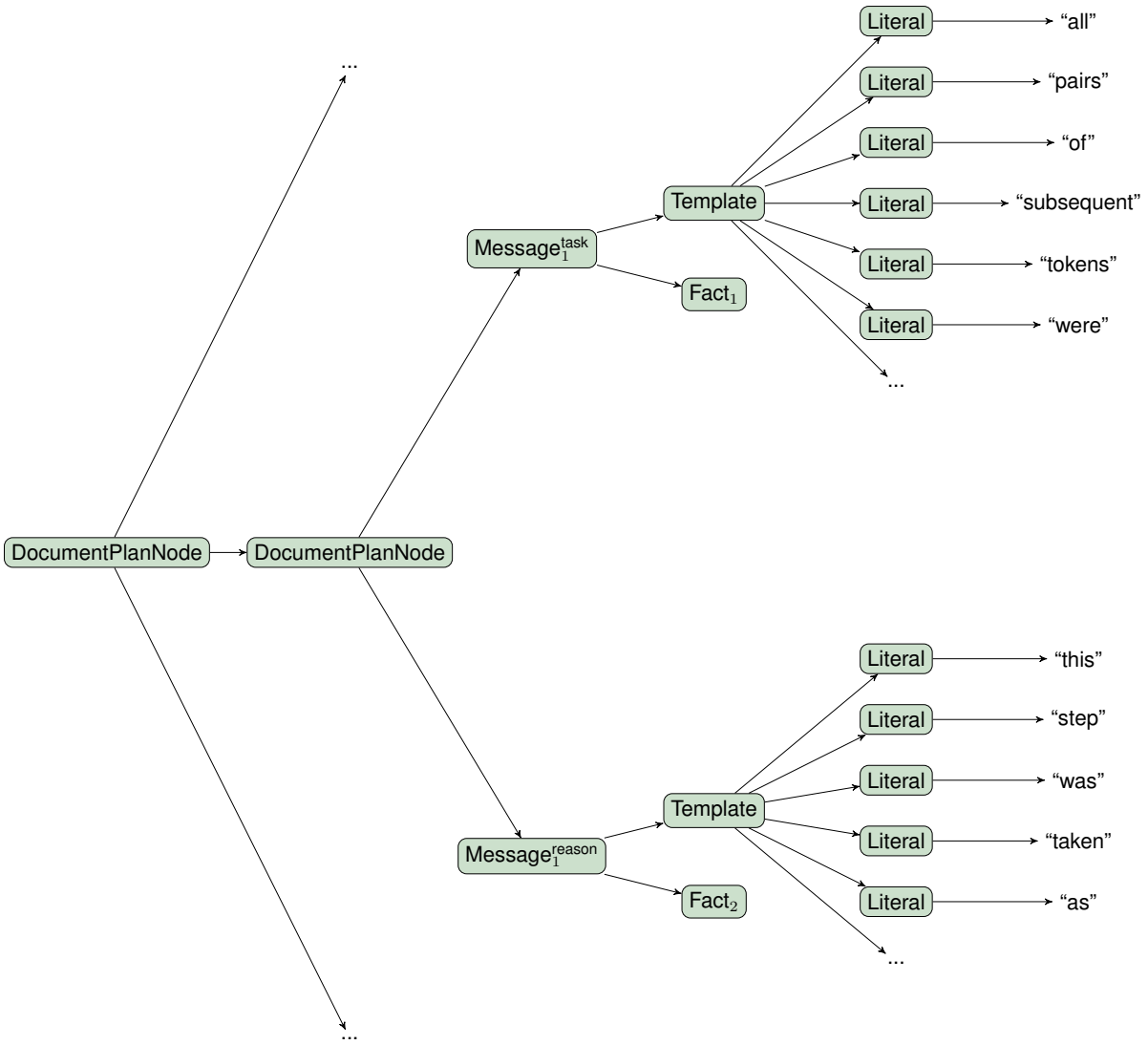

Figure 9: Part of a Document Plan after Lexicalization.

the cost of a significantly more complex syntax – most of the resolvers do their matching via regular expressions – and are thus not a panacea that would also replace the templates.

The lexicalization process is highly dependent on the details of message generation (rather, the individual task and reason parsers – see Section 5.1.3). For each slotted value in a template, the lexicalization process must be able to produce a natural language expression that describes said slotted value. The lexical resolvers are thus integrated to the system the same way the parsers are. They are illustrated in Figure 3 as the small boxes labeled 'Lexical Resolvers'. As they are associated with parsers, and thus individual analyses and heuristics, they too are included in the same per-task and per-heuristic logical components.

After lexicalization, the document plan is a tree akin to that shown in Figure 9. This modified document plan is then passed to the next component in the pipeline.

5.5. Aggregation and Referring Expression Generation

It is at this point in the pipeline that the Reporter (see Deliverable D5.7) conducts processes known as aggregation and referring expression generation. During aggregation, phrases are combined into more complex sentences to improve the fluency of the output and to reduce repetition. During referring expression generation, the system determines how it should refer to various domain entities.

In constructing the Explainer, we determined it has no need for a dedicated Aggregation component: the sentences produced during the templating and lexicalization stages are already so complex that aggregation would not improve the fluency of the text. The structure of the document is also such that we have not been able to identify significant lexical repetition in subsequent sentences.

At the same time, the Explainer's architecture explicitly allows for an aggregation component to be included if the need for one is identified in the future. In fact, due to the Explainer being based on the Reporter, it should be possible to largely reuse the aggregation component from the Reporter with minor changes. These present versions of this component, as it is used in the Reporter, is described in Deliverable D5.7.

Similarly, the Explainer has significantly less need to refer to domain entities than the Reporter. In fact, the only domain entities being referenced by the present version of the Explainer are languages. As such, the entity name resolution component of the Explainer is significantly simplified from that used in the Reporter. At the same time, we have retained the overall design based on Named Entity Resolvers, thus enabling this feature of the system to be extended in the future if necessary. At the present, the only named entity resolver used is one for realizing the names of the languages.

5.6. Morphological Realization

While the English language running example we have been using is very close to correct language, the processing pipeline is not yet complete. Especially in cases of more complex morphology, certain words in the document plan still need to be inflected into their right morphological forms. In English, this is relatively straightforward. However, in other languages such as Finnish, the morphological realization process is significantly more complex.

During morphological realization, each token of the document plan is inspected individually. If it contains morphological information, such as a 'case' attribute, the token and the case are handed to a language specific morphological realizer (shown in Figure 3) which correctly inflects the token.

We have implemented realization systems for both English and Finnish using the UralicNLP library [26]. The English language realization component is at this point unused, as the English morphology is relatively simple and the little inflection necessary (mostly with regard to number) was simpler to implement on the template level. Similarly, we have found no need for French or German morphological tools. If, in the future, need for German or French morphological realization is needed, similar 3rd party tools should be integrated into the Explainer for said languages.

5.7. Realization into HTML

Finally, the document plan, which has been modified by all the previous components, is given to the surface realizer to be realized into a format that can be displayed to the end user. As seen in Figure 9,

- The 10 most salient named entities were identified from the corpus. This was done as part of initializing a new experiment.
- The change of sentiment towards identified entities over time was identified. This step was taken because the preceding step found highly interesting results which the Investigator wants to expand upon.

- The dataset was split by different values of the 'NEWSPAPER_NAME' facets. This action was taken because the original collection was relatively large.
- The publication years, newspapers names and languages of the documents were extracted. This step was taken because the Investigator had previously built a new collection of documents and wants to begin analyzing it.

Figure 10: Extracts of Explainer outputs.

the document plan at the onset of surface realization is in a state where natural language expressions can be formed by traversing the leaves of the tree.

This traversal does not yet, however, completely flatten the tree structure. Rather, it retains the higher-level nodes of the document plan so that, for example, the paragraph structure is not lost. At the same time, the surface realizer adds typographical details such as capitalizing the first word of each sentence and adding sentence-final punctuation.

The final stage of the surface realization is the production of a flat text representation which incorporates any necessary Hypertext Markup Language (HTML) tags. Currently, the Explainer can produce three types of different HTML structures: text paragraphs such as found in standard text documents, list structures where the individual sentences are presented as bullet points and enumerated lists.

Extracts of Explainer outputs are shown in Figure 10.

6. Integration with Assistant Control

The Explainer is integrated to the other NewsEye components in three ways. First, it communicates with the Controller of the Personal Research Assistant, and through it, the Investigator. Second, the Explainer contains functionalities that enable it to describe the tools applied by the Investigator. Third, it is able to describe the internal reasoning, or the heuristics used by the Investigator.

We have described in Section 2.2 the relation of the NewsEye Explainer (i.e. the system described in this Document) to the rest of the NewsEye Personal Research Assistant (Work Package 5), the NewsEye User Interface (Work Package 7) and other components produced by Work Packages 3 and 4 in broad terms. In practice, to enable the Assistant Controller to start a generation task, the Explainer offers a simple Application Programming Interface (API) with three endpoints.

The first endpoint, `/api/languages`, simply reports the languages supported by the Explainer. This enables the technical users (such as the NewsEye Demonstrator user interface) to dynamically display a list of available explanation languages to the end user. The second endpoint, `/api/formats`, returns the set of valid body formatting options. The final endpoint, `/api/explanation` is used to generate new explanations based on ordered list of task-reason pairs given as a parameter. Both API endpoints produce responses in the industry standard format, JSON. The Explainer's API is described in detail, with example responses, in Appendix A. It is in most ways identical to the Reporter API, as the needs of the systems on this front are so well aligned.

7. Integration for Tasks and Reasons

While the Explainer is not directly connected to the tasks completed by the Investigator, it nevertheless needs some tailoring for each of them. Without this tailoring, the Explainer cannot know how to describe the task that was completed. As described in various parts of Section 5, this tailoring needs to occur on three different levels: message generation, templates and lexicalization.

First, the message generator (see Section 5.1.3) needs to be tailored for each type of task so that the task part of the task-reason pairs can be correctly translated to the fact data structures (see Section 5.1.1). This tailoring is provided in the format of *Python 3* code.

Second, it must be ensured that each type of task has a set of templates (see Section 5.3) that allow them to be expressed in natural language. These templates need to be provided in the templating language employed by the Explainer. Of the three types of tailoring, this is the simplest as the new templates can be simply appended to the template database.

Third, the lexicalization component (see Section 5.4) must be tailored so that all concepts embedded into the templates can be expressed in natural language. Since these concepts are defined by the message generator's tailoring, corresponding tailoring needs to be made into the lexicalization component. Like the message generator tailoring, this tailoring is also provided as Python 3 code.

The same elements, in identical formats, need also to be provided for each base heuristic built into the Investigator, so that the reason components of the task-reason pairs can be generated, templatized and lexicalized.

In the message generator and lexicalization cases, the Python 3 code that does the tailoring is provided as a separate Python module that provides a simple API of two functions. The first function takes as input the JSON description of the Investigator's task-reason pairs and return a list of messages (each containing a single fact). The message generator then aggregates the lists received from the different modules into a complete set that is provided to document planner. The second function conducts partial lexicalization and both its input and output are a document plan. The function is expected to modify the tree by lexicalizing all slots it can. By iteratively working the document plan through all the modules, together with some general lexicalization conducted by the lexicalizer itself, the resulting document plan is expected to be completely lexicalized.

8. Evaluation

To evaluate the Explainer, we must consider its success on two primary axes. First, we must consider the crucial aspect of whether it is *fit for purpose*, i.e. whether the texts it produces are useful for the end-users of the Assistant, and whether the Explainer fulfills the need it was supposed to. Second, we must consider the various technical aspects of the Explainer against the design requirements of the system. These requirements are not so much about the *present* state of the Explainer, but rather about how well it is suited for future development. We will consider both of these aspects in turn.

Assessment of the usability/fitness of NewsEye tools for research purpose is the topic of Task T6.1 in WP6. In that task, the Personal Research Assistant and its Explainer component have been tested by the DH group, i.e., by the actual expert historians and humanities researches themselves. Please see Deliverable D6.9, 'Usability/Fit for research purpose test of tools and user interfaces (c)' (26 Feb 2021). In their report, the DH group viewed the Explainer as 'a valuable help, especially when they retrace the steps of their own searches and the PRA's automated experiments'. Following further improvements to the Explainer, in more recent project-internal feedback sessions the historians have been similarly positive about the performance of the tool, describing it as 'very helpful.' We conclude that the Explainer fulfills the intended design objective.

This is, however, not to say that further improvements would not be possible. For one, some of the internal feedback called for even more detail in the explanations. We believe this request exemplifies the idea that the texts produced by the Explainer would benefit from tailoring to individual users: for highly technical users, it would be beneficial to describe in great detail e.g. the parameters of the topic models used, while such information would be simply linguistic noise for users without a technical background. We note that similar observations were made in the context of the Reporter (see Deliverable D5.7). We intend to continue to elicit and address feedback to fine-tune the level of detail in the Explainer outputs to the end of the project.

In terms of technical suitability for continued development, we turn to the requirements analysis described in Section 4. In that analysis, we identified that the system has high requirements for *correctness*, *extensibility* and *multilinguality*, as well as a little need for high variation.

In terms of correctness, we believe the rule-based design of the Explainer fulfills the requirements excellently. The system conducts no statistical (or otherwise) black-box processing and as a result we are satisfied that the system's output always faithfully reflects the system input. If we were to identify programming errors that caused this to not be true in the future, the rule-based approach employed herein allows us to surgically address the issues without having to fear that the modifications would have introduced some completely unexpected and unrelated behaviour problems, as would be possible in the case of e.g. retraining neural models.

The modular design of the system makes it highly extensible. In fact, the present system was constructed in an iterative manner, introducing the various processors one-by-one. Thus, the process of creating the system itself has been a simulation of extending the system to be able to discuss new tasks and heuristics as employed by the Investigator.

Similarly, in terms of multilinguality, the process of creating the system itself was used to validate that the design of the system fulfills the multilinguality requirement. This was done by first implementing the system in English, followed by extensions to Finnish, German and French. The success of these

extensions itself indicates that the fundamental system design is, indeed, multilingual at least within the context of the majority of European languages. The combination of English, French and German provides a good coverage of Indo-European languages, while the inclusion of Finnish indicates that the system is also extensible to non-Indo-European languages as well as languages with significant morphological variation. While this does not preclude a possibility that the architecture makes some fundamental assumptions that would be incompatible with other languages we are not familiar with, we believe the Explainer fulfills the multilinguality requirement well at least within the context of European languages.

As with the Reporter (see Deliverable D5.7), in extending the system both with new languages and to support new Investigator tasks and heuristics we identified that most of the extension time was spent on sourcing translations from native-speaking domain experts. For languages with low morphological complexity (e.g. English), this was by far the most time-costly part of the extension process. For languages with significant morphological complexity (e.g. Finnish), the amount of effort needed is dependent on whether there are any available third party morphological realization libraries. In the case of Finnish, we were able to use the UralicNLP library, making this process relatively simple. However, if the system were to be extended to support a completely new language with high morphological complexity and *without* any available morphological realization libraries, a significant amount of effort would be needed to produce the necessary tooling for morphological realization. At the same time, practically speaking, the system will likely need only a relatively limited coverage of the language's total morphology. For example, in the case of Finnish, we only employ one or two cases from the 14 to 16 present in the Finnish language, and even those in very limited contexts. As such, these considerations do not alter our view that the system design fulfills both the requirements for extensibility and multilinguality well.

9. Conclusions

This report has described the Explainer component of the NewsEye Personal Research Assistant and how it relates to the rest of the NewsEye project and especially the other components of the Personal Research Assistant. The Explainer follows a modularized pipeline architecture for data-to-text NLG. Analyzed in terms of both fitness for purpose and the system requirements identified in Section 4, we believe the architecture described in Section 5 fulfills both well.

Overall, we believe that the Explainer is a successful application of NLG techniques to an extremely challenging domain and application context. We view it as fulfilling both the technical and practical requirements set for the system well. As noted above, the Explainer shares large portions of its code with the Reporter (see Deliverable D5.7). As such, we believe that any future work undertaken on the Reporter will translate to improved linguistic capabilities in the Explainer without significant additional development work.

References

[1] Ross Turner, Somayajulu Sripada, Ehud Reiter, and Ian P Davy. "Using spatial reference frames to generate grounded textual summaries of georeferenced data". In: *Proceedings of the fifth international natural language generation conference*. Association for Computational Linguistics. 2008, pp. 16–24.

[2] François Portet, Ehud Reiter, Albert Gatt, Jim Hunter, Somayajulu Sripada, Yvonne Freer, and Cindy Sykes. "Automatic generation of textual summaries from neonatal intensive care data". In: *Artificial Intelligence* 173.7-8 (2009), pp. 789–816.

[3] Catalina Hallett and Donia Scott. "Structural variation in generated health reports". In: *Proceedings of the Third International Workshop on Paraphrasing (IWP2005)*. Association for Computational Linguistics. 2005, pp. 33–40. URL: http://aclweb.org/anthology/I05-5005.

[4] Ehud Reiter, Somayajulu Sripada, Jim Hunter, Jin Yu, and Ian Davy. "Choosing words in computer-generated weather forecasts". In: *Artificial Intelligence* 167.1-2 (2005), pp. 137–169.

[5] Jose Coch. "Multimeteo: multilingual production of weather forecasts". In: *ELRA Newsletter* 3.2 (1998).

[6] Eli Goldberg, Norbert Driedger, and Richard I Kittredge. "Using natural-language processing to produce weather forecasts". In: *IEEE Expert* 9.2 (1994), pp. 45–53.

[7] Lidija Iordanskaja, Myunghee Kim, Richard Kittredge, Benoit Lavoie, and Alain Polguere. "Generation of extended bilingual statistical reports". In: *Proceedings of the 14th conference on Computational linguistics-Volume 3*. Association for Computational Linguistics. 1992, pp. 1019–1023.

[8] Dietmar Rösner. *The automated news agency: SEMTEX–a text generator for German*. Martinus Nijhoff Publishers, 1987.

[9] Henry H. Eckerson. *The Ultimate Guide to Natural Language Generation Definitions, Trends, and Products*. 2017.

[10] Stefanie Sirén-Heikel, Leo Leppänen, Carl-Gustav Lindén, and Asta Bäck. "Unboxing news automation: Exploring imagined affordances of automation in news journalism". In: *Nordic Journal of Media Studies* 1.1 (2019), pp. 47–66.

[11] Ehud Reiter and Robert Dale. *Building natural language generation systems*. Studies in Natural Language Processing. Cambridge University Press. 2000.

[12] Albert Gatt and Emiel Krahmer. "Survey of the State of the Art in Natural Language Generation: Core tasks, applications and evaluation". In: *Journal of Artificial Intelligence Research* 61 (2018), pp. 65–170.

[13] Dimitra Gkatzia. "Content Selection in Data-to-Text Systems: A Survey". In: *arXiv preprint* (2016). Available at https://arxiv.org/abs/1610.08375.

[14] Anja Belz and Eric Kow. "Extracting parallel fragments from comparable corpora for data-to-text generation". In: *Proceedings of the 6th International Natural Language Generation Conference*. Association for Computational Linguistics. 2010, pp. 167–171.

[15] Nina Dethlefs. "Context-Sensitive Natural Language Generation: From Knowledge-Driven to Data-Driven Techniques". In: *Language and Linguistics Compass* 8.3 (2014), pp. 99–115.

[16] Ondřej Dušek, Jekaterina Novikova, and Verena Rieser. "Findings of the E2E NLG challenge". In: *arXiv preprint arXiv:1810.01170* (2018).

[17] Ehud Reiter. *Hallucination in Neural NLG*. https://ehudreiter.com/2018/11/12/hallucination-in-neural-nlg/. Accessed: 2020-03-02. 2018.

[18] Chia-Wei Liu, Ryan Lowe, Iulian V Serban, Michael Noseworthy, Laurent Charlin, and Joelle Pineau. "How not to evaluate your dialogue system: An empirical study of unsupervised evaluation metrics for dialogue response generation". In: *arXiv preprint arXiv:1603.08023* (2016).

[19] Ehud Reiter and Anja Belz. "An investigation into the validity of some metrics for automatically evaluating natural language generation systems". In: *Computational Linguistics* 35.4 (2009), pp. 529–558.

[20] Feng Nie, Jin-Ge Yao, Jinpeng Wang, Rong Pan, and Chin-Yew Lin. "A simple recipe towards reducing hallucination in neural surface realisation". In: *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*. 2019, pp. 2673–2679.

[21] Ondřej Dušek, David M Howcroft, and Verena Rieser. "Semantic Noise Matters for Neural Natural Language Generation". In: *Proceedings of the 12th International Conference on Natural Language Generation*. 2019, pp. 421–426.

[22] Ratish Puduppully, Li Dong, and Mirella Lapata. "Data-to-text generation with content selection and planning". In: *Proceedings of the AAAI conference on artificial intelligence*. Vol. 33. 01. 2019, pp. 6908–6915.

[23] Ratish Puduppully and Mirella Lapata. "Data-to-text Generation with Macro Planning". In: *arXiv preprint arXiv:2102.02723* (2021).

[24] Thiago Castro Ferreira, Chris van der Lee, Emiel van Miltenburg, and Emiel Krahmer. "Neural data-to-text generation: A comparison between pipeline and end-to-end architectures". In: *arXiv preprint arXiv:1908.09022* (2019).

[25] Leo Leppänen, Myriam Munezero, Mark Granroth-Wilding, and Hannu Toivonen. "Data-Driven News Generation for Automated Journalism". In: *Proceedings of the 10th International Conference on Natural Language Generation*. 2017, pp. 188–197.

[26] Mika Hämäläinen. "UralicNLP: An NLP Library for Uralic Languages". In: *Journal of Open Source Software* 4.37 (2019), p. 1345. DOI: 10.21105/joss.01345.

A. Explainer API Description

A.1. Endpoints

- `GET` /**api**/**languages** - List supported languages
- `GET` /**api**/**formats** - List supported formats
- `POST` /**api**/**explanation** - Produce an explanation

A.2. `GET` /api/languages

Describes the languages supported by the Explainer. All languages in the response are valid to be used as the `language` parameter in the `POST` /**api**/**explanation** request.

Parameters

None

Example Response

```
{
  "languages": [
    "en"
  ]
}
```

A.3. `GET` /api/formats

Describes the text formatting options supported by the Explainer. All formats in the response are valid to be used as the `format` parameter in the `POST` /**api**/**explanation** request.

Parameters

None

Example Response

```
{
  "formats": [
    "p",
    "ol",
    "ul"
```

```
  ]
}
```

A.4. POST /api/explanation

Produces a natural language explanation from a sequence of task-reason pairs. The response consists of two mandatory fields: `language` which describes the language of the explanation and `body` which contains the body text of the explanation as HTML.

The response can also contain an additional `errors` field, which describes any errors encountered during the generation process..

Parameters

Field	Description
`language`	The language the explanation should be written in. Valid values are those returned by the `GET` **/api/languages** endpoint.
`format`	The format of the body of the explanation. Valid values are those returned by `GET` **/api/formats** endpoint. Currently supported values are '`p`' for paragraphs of text, '`ul`' for a list of bullet points and '`ol`' for a list with numbered elements.
`data`	A sequence of task-reason pairs, as provided by the Investigator as a JSON object.

Example Response

```
{
  "language": "en",
  "body": "<p>...</p>",
}
```