Semantic Assistants – User-Centric Natural Language Processing Services for Desktop Clients

René Witte[1] and Thomas Gitzinger[2]

[1] Department of Computer Science and Software Engineering
Concordia University, Montréal, Canada
[2] Institute for Program Structures and Data Organization (IPD)
University of Karlsruhe, Germany

Abstract. Today's knowledge workers have to spend a large amount of time and manual effort on creating, analyzing, and modifying textual content. While more advanced semantically-oriented analysis techniques have been developed in recent years, they have not yet found their way into commonly used desktop clients, be they generic (e.g., word processors, email clients) or domain-specific (e.g., software IDEs, biological tools). Instead of forcing the user to leave his current context and use an external application, we propose a "Semantic Assistants" approach, where semantic analysis services relevant for the user's current task are offered directly within a desktop application. Our approach relies on an OWL ontology model for context and service information and integrates external natural language processing (NLP) pipelines through W3C Web services.

1 Introduction

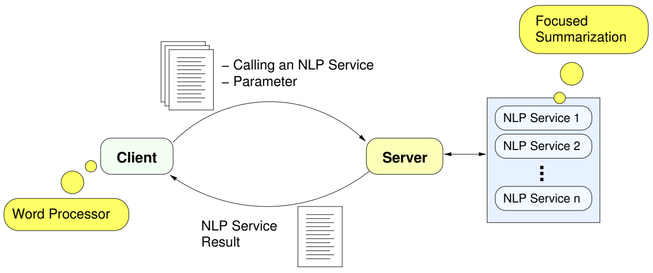
Consider the following scenarios: (1) A scientific journalist, while writing an article on the global climate change, needs to find information on the role of DMSP[1] in the Atlantic marine biology. A *Google* search finds thousands of hits on this topic, forcing our user to interrupt his writing in order to manually evaluate the results. (2) A software developer, while editing code in his IDE, needs to trace a method back to the requirements document in order to understand why a certain feature was implemented. But requirements are not directly linked at the source code level, forcing our developer to interrupt her code analysis and switch to document retrieval and editing tools.

Both scenarios highlight two particularities of today's desktop environments: First, whenever dealing with textual information, users do not get any semantic analysis support besides full-text information retrieval (and document search). Although research in natural language processing (NLP) and text mining has developed a large number of tools and applications within the last decade, like

[1] DMSP stands for "Dimethylsulfoniopropionate" and is a component of the organic sulfur cycle.

machine translation, question-answering, summarization, topic detection, cluster analysis, and information extraction [1], none of these newly developed technologies have materialized in the standard desktop tools commonly used by today's knowledge workers—such as email clients, software development environments (IDEs), or word processors. This directly leads to the second observation: The vast majority of users still relies on manual retrieval of relevant information through an information retrieval tool or website and subsequent manual processing of the (often millions of) results—forcing the user to interrupt his workflow by leaving his current client and performing all the "natural language processing" himself, before returning to his actual task.

The core idea of our *Semantic Assistants* approach is to take the existing NLP frameworks, wrap concrete analysis pipelines in an OWL-based semantic description that can be brokered through a service-oriented architecture (SOA), and allow desktop clients to connect to this architecture with a plug-in interface using Web services. The semantic analysis services, like question-answering or summarization, then become available directly in the desktop tool used for manipulating content. The following figure illustrates this idea:



The user is not concerned with the implementation or integration of these services, from his point of view he only sees context-sensitive Semantic Assistants relevant for his task at hand.

2 Requirements Analysis

In this section, we define the requirements for our *Semantic Assistants* more precisely. As stated before, the central goal is to bring semantic text analysis support to the end user, by integrating NLP systems with desktop clients:

REQUIREMENT #0: SEMANTIC ASSISTANTS ARCHITECTURE. *Provide the infrastructure for bringing semantic services for content analysis and development offered by NLP systems directly to end-user clients in a context-sensitive fashion.*

An immediate observation is that a number of different user groups will be involved in the creation of these assistants: language engineers, software engineers, and end users. Hence, we adopt a *separation of concerns:* First, there is the role of the *end user*. Considering his perspective, we think about a user working with a client program enriched through semantic services, and not knowing or caring much about the underlying technology. Secondly, we adopt the perspective of the *system integrator*. His role consists of "plugging" new clients or NLP services into our architecture. System integrators form an important user group, because with every client and every NLP service, they add value to the overall system. Also, this group's requirements can be quite different from those of the end user group. Additionally, we define the role of the *language engineer*, who is actually creating and developing the semantic services, like question-answering, summarization, or information extraction. However, within this paper we do not discuss the engineering of specific language services—a topic already widely discussed in the literature[1]—since for the integration capabilities of our approach it should be of no concern *how* a language service was created. Likewise, the language engineer should not need to care about whether his NLP service will be integrated or not. Therefore, within this paper, we simply assume that NLP services exist, and do not inquire where they came from. Finally, we assume the *system* perspective, where we take care of properties desirable for the system as a whole.

2.1 End User Requirements

"End users" is a very broad term, and necessarily so: we want to facilitate the work of knowledge workers who use any kind of desktop client while working with textual content. Word processors, email clients, PIMs and IDEs are used by a large variety of users for numerous tasks. There should be no *a priori* assumptions; End user characters can range from secretaries to researchers to school kids to housewives. In particular, this means that we cannot expect these users to have any expertise in semantic technologies, language engineering, or software engineering. The most important and obvious consequence from this observation is the requirement to design a client-independent system architecture:

Requirement #1: Client Independence. *The architecture must be open and flexible with respect to the type of clients integrated with it.*

Thus, the architecture must not be limited to any particular user group or software type. It should even allow the connection of future clients that do not currently exist.

Requirement #2: Context Sensitivity. *Not every analysis method is suitable for every situation, user, or document, but can rather depend on language capabilities (of both the users and analysis services), data formats, and the user's current goal and task.*

In other words, Semantic Assistants must be equipped with some kind of context model that captures the user's context and matches it with applicable services in order to be able to recommend helpful assistants.

2.2 System Integrator Requirements

The second group of users are *system integrators:* developers who integrate either some client software or a new semantic service into our architecture. We want to make their job as easy as possible by taking care of some important issues:

Requirement #3: Facilitate Client Integration. *Clients in the form of end-user applications or user agents partly acting on their own are the user's* *"entry door" to our architecture. Without such clients, the architecture is rather useless, so Requirement #3 is a very fundamental one: Allow for the integration of any client, and enable developers to do this in an effective and efficient manner.*

After all, the main reason current desktop clients do not offer sophisticated semantic NLP services is that their integration involves taking care of many cumbersome details, like connection settings and remote procedure calls. Further refining this requirement, we have to take care of two details: *(1) Server Abstraction:* the integrator must be shielded from low-level communication details between the client and the server. *(2) Implementation of Common Client Functionality:* Many clients will most likely have to fulfill some of the same or similar tasks, like finding available NLP services, passing text from the desktop client, or retrieving results from the service. To avoid duplication of these common functions, a reusable abstraction layer should be provided to system integrators.

REQUIREMENT #4: FACILITATE NLP SERVICE INTEGRATION. *NLP services must not be hardcoded but rather discovered on-the-fly by the architecture.*

This requirement allows for new services to be plugged into the architecture, thereby becoming immediately available to any connected end-user client. In order words, this requires the development of service metadata, based on a formal specification.

2.3 System Requirements

Finally, we address some additional core architectural requirements that stem from the point of view of the "system role." First, it is quite plausible that clients need to have a way to provide input to the semantic services they wish to use—otherwise, these services would have nothing to work on:

REQUIREMENT #5: INTUITIVE INPUT PASSING FOR CLIENTS. *Allow for an easy transfer of unstructured data from the client to the analysis service.*

Language services might require individual parameters, such as the length of a summary to be generated. Some of these parameters can be automatically determined by the architecture (such as input/output connections), but some should be configurable by the end user. Therefore, we have:

REQUIREMENT #6: CONTROL OF LANGUAGE SERVICES. *Provide a way to transmit parameter values from the client to the actual language processing component(s).*

In particular, the architecture must enable parameter detection, client notification of required parameters, and handle correct value assignment of individual parameters to language services (e.g., the length of a summary to be generated).

REQUIREMENT #7: FLEXIBLE RESULT HANDLING. *The final step in the process of invoking a language service includes taking the result or results of the service, wrapping it up in a response message and sending that message back to the client.*

Here, the architecture must be capable of detecting the output(s) provided by the language services: these can be new documents, stored in a file or database, or annotations attached to an existing or newly retrieved document. Our architecture must therefore allow for a description of these outputs along with the description of a language service. This description has to be detailed enough so that the architecture knows how to capture and handle the produced output. Furthermore, to allow for a simple client-side integration, we postulate a uniform response format that must be expressive enough to capture all the various result forms.

2.4 Related Work

Some previous work exists in building personalized information retrieval agents, e.g., for the Emacs text editor [2] or Microsoft Word [3]. These approaches are typically focused on a particular kind of application (e.g., emails or word processing), whereas our approach is general enough to define NLP services independent from the end-user application through an open, client/server infrastructure.

The most widely found approach for bringing NLP to an end user is the development of a new interface (be it Web-based or a "fat client"). These applications, in turn, embed NLP frameworks for their analysis tasks, which can be achieved through the APIs offered by frameworks such as GATE [4] or UIMA [5]. The BioRAT system [6] targeted at biologists is an example for such a tool, embedding GATE to offer advanced literature retrieval and analysis services. In contrast with these approaches, we provide a service-oriented architecture to broker any kind of language analysis service in a network-transparent way. Our architecture can just as well be employed on a local PC as it can deliver focused analysis tools from a service provider. For example, a commercial scientific publisher might want to offer a "related work finder" analysis service, similar to the one presented by [7], to scientists writing research papers or proposals.

Recent work has been done in defining Web services for integrating NLP components. In [8], a service-oriented architecture geared towards terminology acquisition is presented. It wraps NLP components as Web services with clearly specified interface definitions and thus allows language engineers to easily create and alter concatenations of such components, also called processing configurations. Their work is complimentary to our approach, since it addresses the composition of NLP components into pipelines, whereas we are concerned with semantic descriptions of existing pipelines and their integration with desktop clients.

Close in spirit to our approach is the work performed by the *Semantic Desktop* community [9]. The architectures developed in this area, like IRIS [10],[2] aim at deriving, maintaining, and exchanging semantic metadata between desktop applications to provide better semantic support for end users. This work can again be seen as complimentary to our approach, as the type of services offered and their data granularity differs greatly.

[2] The same holds for similar approaches, like Nepomuk, Gnowsis, or Haystack.

3 Semantic Assistants Design

We now describe the design of our Semantic Assistants, in particular the overall system architecture (Section 3.1) and our ontology-based NLP service and context model (Section 3.2).

3.1 System Architecture

An overview of our system architecture developed for the stated requirements is shown in Fig. 1. It is based on a typical multi-tier information system design.

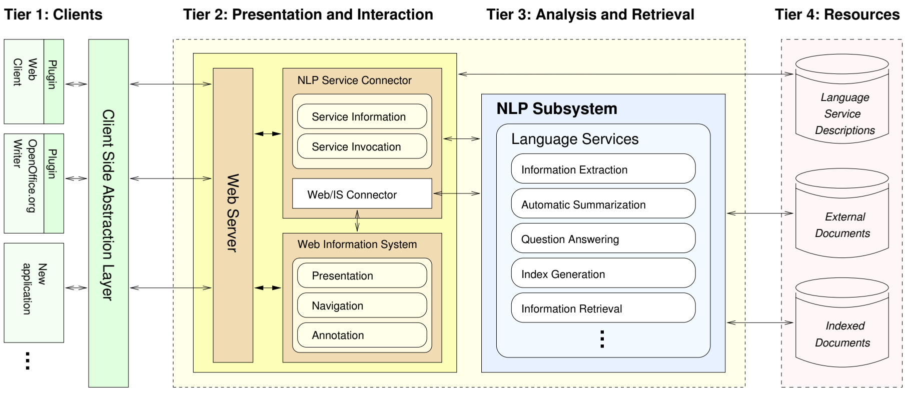


Fig. 1. Architecture for integrating text analysis services and end-user clients

Tier 1: Clients. This tier has the main purpose of providing access to the system. Typically, this will be an existing client (like a word processor or email client), extended to connect with our architecture through a plug-in interface. Besides facilitating access to the whole system, clients are also in part responsible for presentation, e.g., of language service results. In addition to the actual client applications, an abstraction layer is part of Tier 1. It shields the clients from the server and provides common functionality for NLP services, as stipulated by Requirement #3.

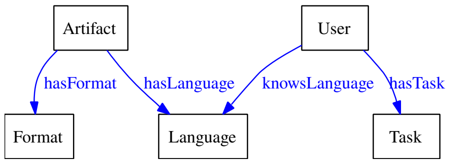
Tier 2: Presentation and Interaction. Tier 2 consists of a standard Web server and a module labeled "NLP Service Connector" in the diagram. One responsibility of this module is interaction, in that it handles the communication flows between the NLP framework and the Web server. Moreover, it prepares language service responses, by collecting results from the NLP services and transforming them into a format suitable for transmission to the client (Requirement #7). Finally, the NLP Service Connector reads the descriptions of the language services, and therefore "knows" the vocabulary used to write these descriptions, as discussed in Requirement #4. In addition, our architecture can also provide services to other (Web-based) information system, like a Wiki [11].

Tier 3: Analysis and Retrieval. Tiers 1 and 2 are the ones the user has direct contact with. Tier 3 is only directly accessed by the NLP Service Connector. It contains the core functionality that we want, through Tiers 1 and 2, to bring to the end user. Here, the semantic services reside, and the NLP subsystem in whose environment they run (such as GATE or UIMA). Language services can be added and removed from here as required.

Tier 4: Resources. The final tier "Resources" contains the descriptions of the language services. These descriptions are read by the NLP Service Connector so that it can satisfy its clients' information needs. Whenever a new language service is added to the architecture, its description must be added here, too (Requirement #4). The vocabulary of these descriptions is based on an OWL ontology using description logics (OWL-DL). Furthermore, indexed documents as well as external documents (e.g., on the Web) count as resources.

3.2 The Semantic Assistants Ontology

As discussed in Requirement #2, in order to be able to recommend semantic services relevant for the user's current task, we need to model the current context, which includes the user's task, available services, the artifacts involved (services, documents, etc.) and their languages. We previously developed an upper ontology for supporting software processes [12], which we adapted for the Semantic Assistants setting. It includes five essential concepts that form the basis of this upper ontology, namely *Artifact*, *Format*, *User*, *Language*, and *Task*:



Artifacts include both content (documents, Web pages, etc.) and tools (such as an NLP tool). Formats (e.g., OWL, XML, or PDF) have been modeled separately from languages (both natural languages and artificial languages) as they are largely orthogonal. Users are modeled with their language capabilites and the tasks they need to perform.

Modeling Artifacts. We can now start to extend the *Artifact* concept of the upper ontology with concepts required for Semantic Assistants (Fig. 2), as stipulated by Requirement #4. We have just mentioned *Tool* as a sub-concept of *Artifact*, a tool possibly processing artifacts as input and producing artifacts as output (*consumesInput* and *producesOutput* relations). Additionally, a tool may require parameters (*hasParameter* relation). For Semantic Assistants, *documents* are a focal point. NLP services are often language-specific, so to be able to only offer assistants relevant for the language of a document a user is working on, and the language(s) he understands, we make use of the *hasLanguage* relation.

In order to work with documents, they often must somehow be identified and retrieved. In particular, the server must be able to pull documents from a networked source, including the Internet. Thus, when we model such an input

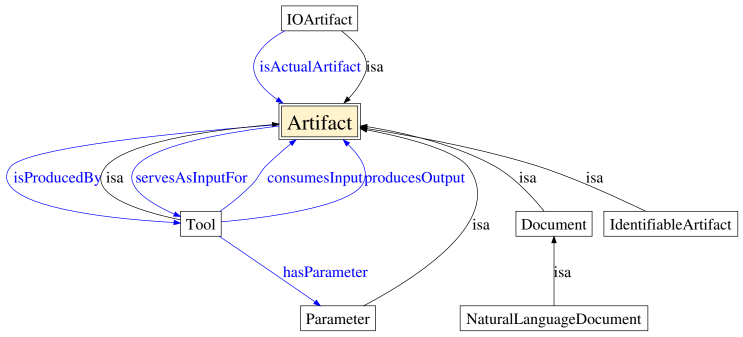


Fig. 2. Modeling Artifacts in the Upper Ontology

document in our ontology, we must have a way to specify its URI (Uniform Resource Identifier) by which we can address it. As not only documents, but also other artifacts like Web services have to be uniquely identified and addressed, we introduce *hasIdentifier* as an optional property for artifacts. We define *IdentifiableArtifact* as a class whose members are artifacts and have this property. In practice, the identifier can, and often will, be a URI, but it does not have to be. For example, if we have a set of elements with unique names, a simple string can be enough. With *hasIdentifier*, we provide an important property on the highest abstraction level, while leaving the exact semantics to the concrete ontologies and the applications that use them.

The same tool can often produce different output (types), depending on both *how* it is invoked (parameters) and the type of the input artifacts (if any). To model this fact, we introduced a concept called *IOArtifact*, where information on input and output relationships can be stored. We will show a more concrete use of this concept in the following section, when we concretize the upper ontology. By means of an *isActualArtifact* relation, we have *IOArtifact* individuals "point" to *Artifact* individuals. They can be seen as proxies for artifacts.

Specializing the Upper Ontology. The upper ontology that we have just introduced provides us with several concepts we need: artifacts, users, parameters, tools, etc. However, to integrate NLP analysis tools on a semantic level, we have to refine this abstract ontology for language services. While the overall design is largely independent from a concrete NLP system, at this point we also introduce concepts specific to our environment, which is based on GATE [4]. However, other NLP subsystems (like UIMA [5]) can be integrated in a similar fashion.

To be able to offer semantic services to end users, we need to model existing NLP analysis services, like summarization (Requirement #4). Towards this end, we introduce new child concepts to the *Tool* concept, classifying language services into two categories: *IRTool* and *NLPTool*. The semantics that we want to convey by this separation is that an information retrieval tool *(IRTool)* finds documents, but leaves them untouched, while an NLP tool processes documents and typically generates some new artifact(s) from them. For NLP tools, input and output natural languages can be specified, if they are language-specific. A *GATEPipeline* is an analysis service with a certain format that can perform either type (or both) of document processing. Instances of *GATEPipeline* are the language services we offer the user through our architecture.

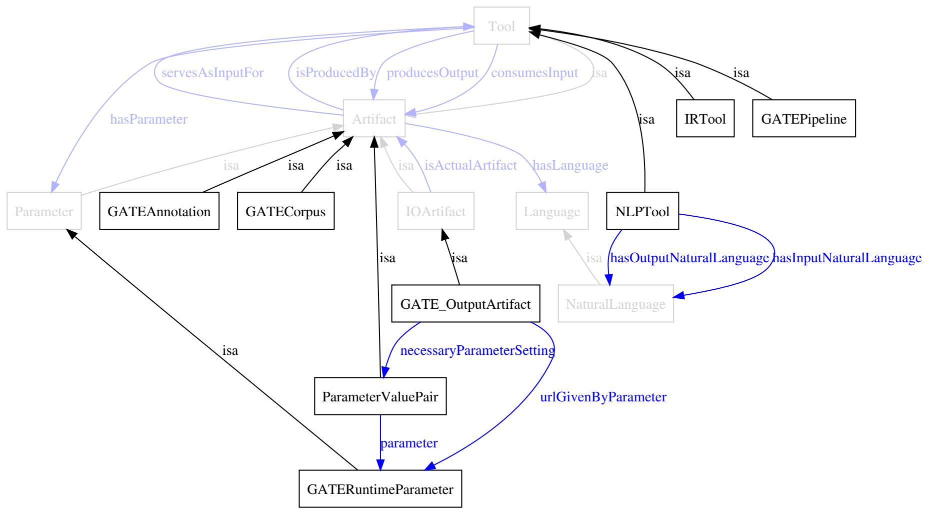


Fig. 3. Additional classes introduced in the specialized ontology

To be able to invoke concrete services, we also have to extend the *Artifact* concept. A *GATECorpus* represents a collection of documents, which typically serves as input for a language service. *GATEAnnotation* instances describe the information that language services add to a document during processing, including their result output. *GATERuntimeParameter* models parameters that control certain aspects of a GATE pipeline, and is introduced as a sub-concept to the already existing *Parameter* concept. Along with these artifact types, we have to introduce corresponding new formats (remember that artifacts are required to have a format). These are defined under the common parent concept of *GATEFormat*, which is a child of the *Format* concept.

We mentioned earlier that output formats can change depending on both parameters passed to a tool (like a GATE pipeline) and its input document, which led to the introduction of the *IOArtifact* concept. To allow our architecture to automatically retrieve the obtained result and deliver it to the end user, we introduce the *GATE_OutputArtifact*. Instances of *GATE_OutputArtifact* have a property *necessaryParameterSetting*, thus connecting an output artifact to a certain parameter value. This parameter value is represented through an instance of a newly introduced concept called *ParameterValuePair*.

4 Implementation

We now discuss selected aspects of the current implementation of our architecture and briefly describe the process of integrating new (desktop) clients in Section 4.5.

4.1 Language Service Description and Management

We start discussing the implementation from the status quo, a common component-based NLP framework—in our case, GATE [4]. The NLP subsystem allows us to load existing language services, provide them with input documents and parameters, run them, and access their results. The GATE framework's API also permits to access all the artifacts involved in this process (documents, grammars, lexicons, ontologies, etc.). Language services take the form of (persistent) *pipelines* or *applications* in GATE, which are composed of several sequential *processing resources* or *PRs*. As mentioned in Section 2, creating a language service (e.g., for summarization or question-answering) is the responsibility of a language engineer, and need not further concern us here.

Our ontology described in the previous section has been implemented using OWL-DL.[3] For each deployed language service, the corresponding entries in the Semantic Assistants ontology need to be created and stored on the server-side (Fig. 1, Tier 4). This permits us to dynamically find, load, parametrize, and execute available language services, based on the user's current task and language capabilities (this is further discussed in Section 4.4). Additionally, the ontology contains all information needed to to locate and retrieve the result(s) delivered (Requirement #7).

4.2 Web Services

Thus far, we can search, load, parametrize, and execute language services. However, all input/output channels are still local to the context of the NLP framework's process. To make NLP services available in a distributed environment, we have to add network capabilities, which we achieve using *Web services*, a standard defined by the W3C:[4] *"A Web service is a software system designed to support interoperable machine-to-machine interaction over a network. It has an interface described in a machine-processable format (specifically WSDL[5]). Other systems interact with the Web service in a manner prescribed by its description using SOAP[6] messages, typically conveyed using HTTP with an XML serialization in conjunction with other Web-related standards."* In essence, a requester agent has to know the description of a Web service to know how to communicate with it, or, more accurately, with the provider agent implementing this Web Service. It can then start to exchange SOAP messages with it in order to make use of the functionality offered by the service. Provider agents are also referred to as Web service *endpoints*. Endpoints are referenceable resources to which Web service messages can be sent. Within our architecture (Fig. 1), the central piece delivering functionality from the NLP framework as a Web service endpoint is the NLP Service Connector. Our implementation makes use of the Web service code generation tools that are part of the Java 6 SDK and the Java API for XML-Based Web services (JAX-WS).[7]

[3] OWL Web Ontology Language Guide, http://www.w3.org/TR/owl-guide/
[4] Web Services Architecture, see http://www.w3.org/TR/ws-arch/
[5] Web Services Description Language (WSDL), see http://www.w3.org/TR/wsdl
[6] Simple Object Access Protocol, see http://www.w3.org/TR/soap/
[7] Java API for XML-Based Web Services (JAX-WS), see https://jax-ws.dev.java.net/

With all necessary artifacts in place, we can now generate and publish the Web service. The JAX-WS API provides convenient functions for this, so that, with two lines of source code (comments not counted), we can start a Web server integrated with the Java environment, and publish the Web service at an address of our choice:

```
// Create SSB instance
SemanticServiceBroker agent = new SemanticServiceBroker();
// Publish SSB instance as Web service endpoint
Endpoint endpoint = Endpoint.publish("http://localhost/...", agent);
```

4.3 The Client-Side Abstraction Layer (CSAL)

We have just published a Web service endpoint, which means that the server of our architecture is in place. On the client side, our *client-side abstraction layer* (CSAL) is responsible for the communication with the server. This CSAL offers the necessary functionality for clients to detect and invoke brokered language services. The implementation essentially provides a proxy object (of class `SemanticServiceBroker`), through which a client can transparently call Web services. A code example, where an application obtains such a proxy object and invokes the `getAvailableServices` method on it to find available language analysis services, is shown below:

```
// Create a factory object
SemanticServiceBrokerService service = new SemanticServiceBrokerService();
// Get a proxy object, which locally represents the service endpoint (= port)
SemanticServiceBroker broker = service.getSemanticServiceBrokerPort();
// Proxy object is ready to use. Get a list of available language services.
ServiceInfoForClientArray sia = broker.getAvailableServices();
```

4.4 Dynamic Assistant Generation

The ontology described in Section 3.2 contains the information needed to dynamically find, load, parametrize, and execute available language services, based on user's current task and language capabilities. In our implementation, it is queried using Jena's SPARQL[8] interface, using the context information delivered by the client plug-in, in order to recommend applicable Semantic Assistants.

For example, when a recommendation request is received, with a context object saying the user knows English and German, the generated SPARQL query should restrict the available services to those that deliver English or German as output language. A simplified version of such a generated query is shown below:

```
SELECT ?x ?name
WHERE { ?x sa:hasGATEName ?name .
       {?x cu:hasFormat sa:GATECorpusPipeline_Format} . {
          {?x sa:hasOutputNaturalLanguage cu:en} UNION
             {?x sa:hasOutputNaturalLanguage cu:de}}
}
```

Once the SPARQL query has been generated, it is passed to the `OntModel` instance containing the language service descriptions. The results are then retrieved from this object, converted into the corresponding client-side versions, and returned to the client.

[8] SPARQL Query Language for RDF, see http://www.w3.org/TR/rdf-sparql-query/

4.5 Client Integration

After describing the individual parts of our architecture's implementation, we now show how they interact from the point of view of a system integrator adding Semantic Assistants to a client application. The technical details depend on the client's implementation: If it is implemented in Java (or offers a Java plug-in framework), it can be connected to our architecture simply by importing the CSAL archive, creating a `SemanticServiceBrokerService` factory, and calling Web services through a generated proxy object. After these steps, a Java-enabled client application can ask for a list of available language services, as well as invoke a selected service. The code examples shown above demonstrate that a developer can quite easily integrate his application with our architecture, without having to worry about performing remote procedure calls or writing network code.

A client application developer who cannot use the CSAL Java archive still has access to the WSDL description of our Web service. If there are automatic client code generation tools available for the programming language of his choice, the developer can use these to create CSAL-like code, which can then be integrated into or imported by his application.

5 Application

In this section, we present a real-world application scenario for our architecture: the integration of Semantic Assistants into a word processor.

5.1 The OpenOffice.org Writer Plug-In

Word processor applications are one of the primary tools of choice for many users when it comes to creating or editing content. Thus, they are an obvious candidate for our approach of bringing advanced NLP support directly to end users in form of Semantic Assistants. We selected the open source OpenOffice.org[9] application *Writer* for integration. With its plug-in framework, extensions can easily be added to any OpenOffice.org application.

Following the steps described in Section 4.5, we developed a Java plug-in for *Writer* that offers the functionality to connect with our architecture, inquire about available language services, offers additional dialogs for selecting services and setting required parameters, and handles passing of input documents and NLP results. Depending on the selected language service, either the full document a user is working on can be sent to the language service, or only a highlighted text segment. On the back-end, we integrated some of the language services we developed for other projects, which include index generation, automatic summarization, and question-answering (focused summarization).

Our plug-in creates a new menu entry "Semantic Assistants," as shown in Fig. 4. In this menu, the user can inquire about available services, which are selected based on the client (here *Writer*) and the available languages, as described in Section 4.4. The dynamically generated list of available services is then

[9] Open source office suite OpenOffice.org, see http://www.openoffice.org/

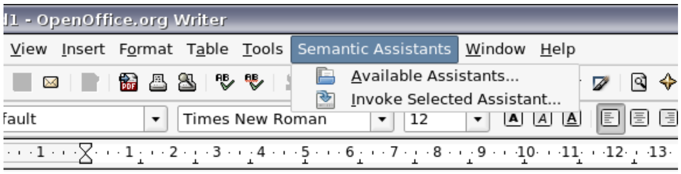

l1 - OpenOffice.org Writer
View Insert Format Table Tools Semantic Assistants Window Help
Available Assistants...
Invoke Selected Assistant...
fault
Times New Roman
12

Fig. 4. “Semantic Assistants” menu entry in OpenOffice.org Writer

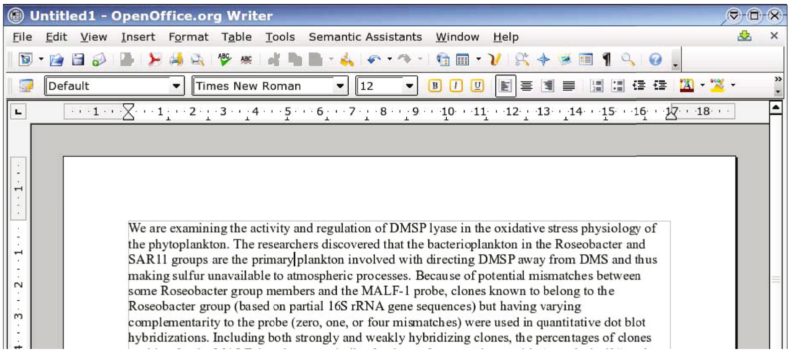
presented to the user, together with a brief description, in a separate window. The user can then select an assistant and execute it. In case the service requires additional parameters, such as the length of a summary to be generated, they are detected by our architecture through the OWL-based service description and requested from the user through an additional dialog window.

Once invoked, the language service is executed asynchronously by our architecture, allowing the user to continue his work (he can even execute additional services). Note that all low-level details of handling language services, such as metadata lookup, parametrization, and result handling, are hidden from the client plug-in through our client-side abstraction layer.

5.2 Example Use Case

One direct use case of our Semantic Assistants is to satisfy information needs of a knowledge worker. As motivated in Section 1, language services can deliver focused analysis results directly within the client—here a word processor—needed to perform a task, rather than interrupting the user's workflow by forcing him to perform an external (Web) search.

Let us go back to the example scenario stated in the introduction: a scientific journalist who is writing a report on the global climate change and needs information on the role of *“DMSP in the Atlantic marine biology.”* Using our Semantic Assistants, he can simply highlight this phrase in the editor and select the “Web Retrieval Summarizer.” This is a compound assistant that performs


Untitled1 - OpenOffice.org Writer
File Edit View Insert Format Table Tools Semantic Assistants Window Help
Default
Times New Roman
12

We are examining the activity and regulation of DMSP lyase in the oxidative stress physiology of the phytoplankton. The researchers discovered that the bacterioplankton in the Roseobacter and SAR11 groups are the primary plankton involved with directing DMSP away from DMS and thus making sulfur unavailable to atmospheric processes. Because of potential mismatches between some Roseobacter group members and the MALF-1 probe, clones known to belong to the Roseobacter group (based on partial 16S rRNA gene sequences) but having varying complementarity to the probe (zero, one, or four mismatches) were used in quantitative dot blot hybridizations. Including both strongly and weakly hybridizing clones, the percentages of clones

Fig. 5. The result of the “Web Retrieval Summarizer” Semantic Assistant, answering the user's question, is presented as a new *Writer* document

two tasks: In a first step, a selected number of hits from a Yahoo! search using the highlighted phrase is retrieved to build a corpus on-the-fly. This corpus is then fed into the multi-document summarizer ERSS [13] to produce a summary answering the question(s) (a so-called *focused summary*). All these actions are performed in the background, allowing the user to continue with other parts of his report. When the summary is ready, the architecture notifies the plug-in, which then presents the generated summary in a new window (Fig. 5). The user can now inspect the result, refine it, and continue with other parts of the report.

6 Conclusions and Future Work

In this paper, we presented the idea of "Semantic Assistants" that aim to help users dealing with the proverbial information overload. In particular, we address end-users that need to find, analyze, or write any kind of textual content. The central idea of our work is that such support should be offered directly integrated into the clients users are accustomed to when working on natural language data: their email clients, Web browsers, word processors or editors. Thus, instead of offering analysis services through a Web interface or custom-build applications, we propose to integrate them directly into end-user clients by means of a service-oriented architecture, based on W3C Web services. An important design goal of our architecture is that it should be as easy as possible to integrate existing clients through plug-in frameworks on the user side, and new semantically-oriented NLP services on the server side.

Our work is the first that aims to bring existing NLP analysis services directly to end users. We believe this is an important goal as there have been numerous advances in the areas of NLP and text mining over the last decade—but none of the newly developed tools have yet found their way into today's desktop environments. Rather, in order to find and process content, users still have to leave their application of choice and perform an external (desktop or Internet) search, forcing a mentally expensive context-switch. In our paradigm, desktop applications can directly react to the user's need for retrieving and analyzing content by offering semantically-oriented services, such as question-answering or summarization, within the same interface. We achieve this by adding a layer of "semantic glue" using an OWL-DL context and service ontology that permits us to connect the existing, but so far separated, worlds of desktop applications and NLP frameworks in a way that brings added value to end users.

Our implementation shows that these ideas can be implemented with current, off-the-shelf tools and open standards. A first practical evaluation of our approach, by integrating a major open source application, the OpenOffice.org *Writer* program, proves that the Semantic Assistants concept can be deployed on a contemporary desktop environment. Future user studies will need to be performed to evaluate the impact of such services on the completion of predefined tasks; however, this will obviously highly depend on the selected type of client, the tasks, and the deployed language services and must therefore not be confused with the evaluation of our architecture and ontology model as such.

Obviously, many extensions are still possible throughout the architecture, but the one most beneficial to end users will be the development of new client plug-ins, bringing further semantic support to, e.g., email clients (searching for relevant information and providing answers to questions), software development environments (linking code to its documentation and offering support when modifying either side), or domain-specific tools (like for biologists or architects).

We believe that bringing the existing, hard-won advances in natural language processing, like question-answering, summarization, or opinion mining, to a larger user base will have significant impact on the fields of NLP, semantic desktop research, and software engineering work. Users can directly benefit from a large number of developed technologies that so far have been limited to expert users and proprietary commercial applications. The developed architecture will be made available under an open source license, which we hope will foster a vibrant ecosphere of client plug-ins.

References

1. Feldman, R., Sanger, J.: The Text Mining Handbook: Advanced Approaches in Analyzing Unstructured Data. Cambridge University Press, Cambridge (2006)
2. Rhodes, B.J., Maes, P.: Just-in-time Information Retrieval Agents. IBM Syst. J. 39(3-4), 685–704 (2000)
3. Colbath, S., Kubala, F.: TAP-XL: An Automated Analyst's Assistant. In: Proc. NAACL 2003, ACL, pp. 7–8 (2003)
4. Cunningham, H., Maynard, D., Bontcheva, K., Tablan, V.: GATE: A Framework and Graphical Development Environment for Robust NLP Tools and Applications. In: Proc. of the 40th Anniversary Meeting of the ACL (2002), http://gate.ac.uk
5. Ferrucci, D., Lally, A.: UIMA: An Architectural Approach to Unstructured Information Processing in the Corporate Research Environment. Natural Language Engineering 10(3-4), 327–348 (2004)
6. Corney, D.P., Buxton, B.F., Langdon, W.B., Jones, D.T.: BioRAT: Extracting Biological Information from Full-Length Papers. Bioinformatics 20(17), 3206–3213 (November 2004)
7. Zeni, N., Kiyavitskaya, N., Mich, L., Mylopoulos, J., Cordy, J.R.: A lightweight approach to semantic annotation of research papers. In: Kedad, Z., Lammari, N., Métais, E., Meziane, F., Rezgui, Y. (eds.) NLDB 2007. LNCS, vol. 4592, pp. 61–72. Springer, Heidelberg (2007)
8. Cerbah, F., Daille, B.: A service oriented architecture for adaptable terminology acquisition. In: Kedad, Z., Lammari, N., Métais, E., Meziane, F., Rezgui, Y. (eds.) NLDB 2007. LNCS, vol. 4592, pp. 420–426. Springer, Heidelberg (2007)
9. Decker, S., Park, J., Quan, D., Sauermann, L. (eds.): Proc. of the 1st Workshop on The Semantic Desktop, Galway, Ireland, CEUR Workshop Proceedings, vol. 175 (November 6, 2005), CEUR-WS.org
10. Cheyer, A., Park, J., Guili, R.: IRIS. Integrate. Relate. Infer. Share. In: [9] (2005)
11. Witte, R., Gitzinger, T.: Connecting Wikis and Natural Language Processing Systems. In: Proc.of the 2007 Intl. Symp. on Wikis, WikiSym 2007 (2007)
12. Rilling, J., Meng, W.J., Witte, R., Charland, P.: A Story Driven Approach to Software Evolution. IET Software (2008)
13. Witte, R., Bergler, S.: Fuzzy clustering for topic analysis and summarization of document collections. In: Kobti, Z., Wu, D. (eds.) Canadian AI 2007. LNCS, vol. 4509, pp. 476–488. Springer, Heidelberg (2007)