Integration of Workflow and Pipeline for Language Service Composition

Mai Xuan Trang, Yohei Murakami, Donghui Lin, and Toru Ishida

Department of Social Informatics, Kyoto University
Yoshida-Honmachi, Sakyo-Ku, Kyoto, 606-8501, Japan
trangmx@ai.soc.i.kyoto-u.ac.jp, {yohei, lindh, ishida}@i.kyoto-u.ac.jp

Abstract

Integrating language resources and language services is a critical part of building natural language processing applications. Service workflow and processing pipeline are two approaches for sharing and combining language resources. Workflow languages focus on expressive power of the languages to describe variety of workflow patterns to meet users' needs. Users can combine those language services in service workflows to meet their requirements. The workflows can be accessible in distributed manner and can be invoked independently of the platforms. However, workflow languages lack of pipelined execution support to improve performance of workflows. Whereas, the processing pipeline provides a straightforward way to create a sequence of linguistic processing to analyze large amounts of text data. It focuses on using pipelined execution and parallel execution to improve throughput of pipelines. However, the resulting pipelines are standalone applications, i.e., software tools that are accessible only via local machine and that can only be run with the processing pipeline platforms. In this paper we propose an integration framework of the two approaches so that each offests the disadvantages of the other. We then present a case study wherein two representative frameworks, the Language Grid and UIMA, are integrated.

Keywords: Service Workflow, Processing pipeline, Language Grid, UIMA

1. Introduction

The creation of language resources (LRs) remains a fundamental activity in the field of language technology. The number of language resources available on the internet has been increasing year by year (Maegaard et al., 2005). Based on these resources, developers can build advanced Natural Language Processing (NLP) applications such as Watson (Ferrucci et al., 2010) and Siri by combining them. However, it is difficult for developers to collect and combine the most suitable set of language resources in order to achieve the intended goals.

Numerous of platforms have been proposed for creating, coordinating and making language resources available and readily useable for users. There are two approaches to sharing and combining language resources: framework-based service workflow such as the Language Grid (Ishida, 2006), and PANACEA (Bel, 2010) and Framework-based processing pipeline such as UIMA (Ferrucci and Lally, 2004b), and GATE (Cunningham et al., 2002). Interoperability is the main challenge to allow different language resources work together in one framework, as well as in different frameworks. Interoperability between components in one framework is dealt by defining common data exchange format between components, or defining standard interface for components. For example, UIMA defines Common Analysis Structure as data exchange between components, the Language Grid defines standard interfaces for their language services in a linguistic service ontology (Hayashi et al., 2011). Interoperability among formats of two processing pipeline frameworks UIMA and GATE is explored in (Ide and Suderman, 2009). This paper addresses the issue of how to bridge the gap between two different common data exchange formats used in different processing pipeline frameworks. Thid work focuses on integrating two different types of frameworks.

The processing pipeline infrastructure provides platforms that define language resources and tools as annotators and combine these annotators in pipeline manner. With support of pipelined execution and parallel execution, this approach has shown a good performance when processing huge amounts of data. After creating applications by combining existing components into pipelines, often users need to share applications that they have developed with other users. To facilitate this, most of processing pipeline platforms offers an import/export mechanism. However, pipelines are normally shareable only within the boundaries of the specific platform. This makes it difficult to use pipelines independently of the platform in which they were developed and violates the principles of wide software applicability and usability. On the other hand, the service workflow infrastructure provides platforms to share language resources as web services and compose these services to define composite services in workflows. Workflow languages have expressive power to describe variety of workflow patterns to satisfy users' requirements. This approach also provides mechanism to share language services of language resources consistent with intellectual property rights. User can access and invoke workflows independently with the platforms. These advantages of service composition platforms encourage language resource providers publish and share their components, increasing the accessibility and usability of language resources. However, transferring huge amounts of data between web services in workflows is not efficient (Zhang et al., 2013).

Therefore, this paper realizes integration of those two types of infrastructures to mutually offset their disadvantages. To this end, we address the following issues:

- Mapping between service invocation interface (service composition) and stand-off annotation (processing pipeline).

- Integrating service workflow engine and pipeline en-

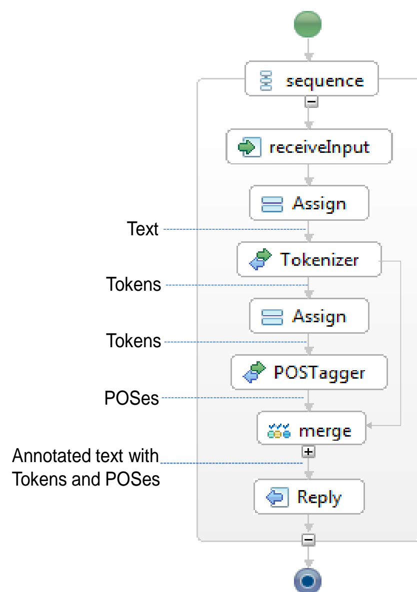


Figure 1: Service composition approach

gine.

- A concept integration framework that can integrate the two types of frameworks.
- To realize the concept framework, a case study of integration two representative frameworks, the Language Grid and UIMA, is described.

The remainder of this paper is organized as follows: Section 2 briefly discusses features of the two types of language resource coordination frameworks. The integration of the service workflow and processing pipeline will be presented in section 3. We show a case study on integration between the Language Grid and UIMA in section 4. A discussion is given in section 5. Finally, section 6 concludes this paper.

2. Language Resource Coordination Frameworks

In this section we briefly discuss about characteristics of two types of language resource coordination frameworks: service composition with interface invocation and processing pipeline with stand-off annotation.

2.1. Service Workflow Approach

This approach takes the Service-Oriented Architecture (SOA) line; language resources are wrapped as web services that users can combine to create customized composite language services as needed. Figure 1 shows a composite service consisting of two language services: Tokenizer and POSTagger. Each service in the workflow is defined by an interface with input and output. For example, Tokenizer service has plain text inputs and outputs a set of tokens of the plain text. Interoperability between services in a workflow is ensured by conforming interfaces of the services. Output of a previous service and input of the later service in the workflow must be compatible.
To help users easily create workflow, standard interfaces for language services have been proposed. The Language Grid designed interfaces of all services based on a language service ontology (Hayashi et al., 2011), with standardization depending on the types of services. PANACEA also defines common interfaces for different types language services such as translation service, bilingual dictionary, etc..
Language resources available as language services on these frameworks are provided by wide variety of companies, research organizations with different intellectual property rights. PANACEA currently has more than 160 services provided by 11 service providers. Over 170 services available on the Language Grid, these services are provided by over 140 groups from 17 countries. The service composition approach provides access control functionality to deal with the issues on intellectual rights of language resources providers. The providers can configure the permission process and monitor the statistics of use of their resources anytime. This advantage of service composition approach encourages providers to share their language resources, increasing number of language services available on these frameworks.

2.2. Processing Pipeline Approach

This approach focuses on providing a setting for creating analysis pipelines, and is oriented towards linguistic analysis and stand-off annotation. The purpose of these frameworks is to combine language resources to analyze huge amounts of data in the local environment. Pipelined execution and parallel execution are employed in pipeline engine to improve throughput of the pipelines.
Language resources are combined into a pipeline to analyze documents. Each resource is defined as an annotator that tags document with annotations in stand-off manner. The pipeline outputs the input document enriched with annotations added by components in the pipeline. The annotated document is written in Common Data Exchange Format (CDEF). The CDEF document passed along the components in the pipeline. Figure 2 shows a pipeline of three annotators: Tokenizer, POSTagger, and Parser. Each annotator in the pipeline processes input text by annotating it. In this example the input text is given three annotations: Token, POS, and ParseTree. The GATE framework provides GATE document as CDEF to represent text and its annotations, this data structure is exchanged between GATE components. UIMA also defines a CDEF called Common Analysis Structure (CAS) to represent text including annotations as a common data exchange between UIMA components.

3. Integration of Service Composition and Processing Pipeline

The processing pipeline is based on the stand-off annotation model, while service composition is based on service interface invocation model. This section first defines a mapping of the two models, and then uses it to define an integration framework.

3.1. Mapping Service Interface Invocation and Stand-off Annotation

The Common Data Exchange Format (CDEF) plays an important role in helping the components in a pipeline to work together. CDEF data structure is defined based on de-facto

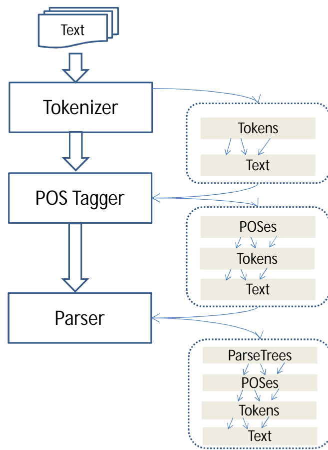


Figure 2: Processing pipeline approach

standards that are used for encoding machine-readable texts and linguistic corpora, such as CES (Ide et al., 2000), TEI (Vanhoutte, 2004), and the *common interface format* being developed under the context of ISO committee TC 37/SC 4 (Ide and Romary, 2009). CDEF basically consists of two parts: one representing the document text, and the other representing annotations. Figure 3 shows an example of CDEF in XML-based format:

- <*doc*>: represents the document, the *id* attribute is used to distinguish documents when a pipeline processing multiple documents. The *docString* attribute represents document text or reference link to the document.

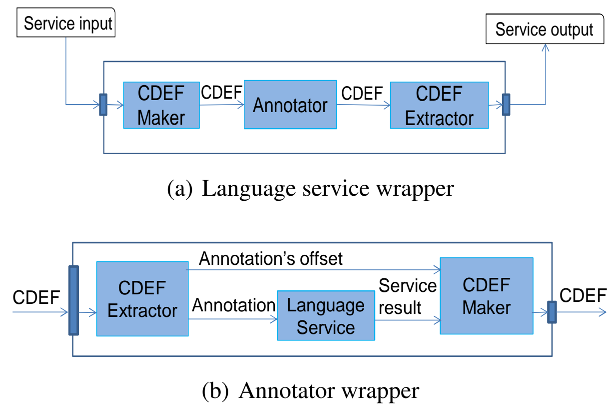
- <*annotations*>: represents all annotations produced by a pipeline. An annotation is described by <*annot*> tag, the *type* attribute indicates type of the annotation, two attributes *begin* and *end* define position of the annotation in the document text and the *componentID* indicates the author(annotator) of this annotation.

- Different annotation types may have different structures. The structure of an annotation type is defined by a feature structure (*fs*) tag. Each feature is defined by a (*f*) tag and its attribute *name* indicates the name of the feature, attribute *value* is the value of the feature. For example, the feature structure of a Morphem produced by a POSTagger consists of two features: lemma and postag of a word, as shown in the example.

In web service composition workflow, each language service has its own interface with input and output. This interface is defined by standard interfaces of the frameworks. The input and output types are designed according to the type of the language service. For example, for a Tokenize service type, input is plain text and output is set of Tokens. In order to interwork two kinds of systems, first we need to map between stand-off annotation model and interface invocation model. For an annotator in processing pipeline, we can assume that its input and output are CDEF documents. The mapping is defined to so as to map input/output of language services to annotation types in CDEF. We define CDEF Maker and CDEF Extractor to conduct the map-

```
<?xml version="1.0" encoding="UTF-8"?>
<annotatedDoc>
  <doc id="1" mimeType="text"
        docString="Text of the document"/>
  <annotations>
     <annot type="POS" docID="1" begin="1"
              end="5" componentID="POSTager">
       <fs>
         <f name="lemma" value="Text"/>
         <f name="postag" value="noun"/>
         ...
       </fs>
     </annot>
     ...
  </annotations>
</annotatedDoc>
```

Figure 3: CDEF Structure



(a) Language service wrapper

(b) Annotator wrapper

Figure 4: Wrappers

ping and create two wrappers: Language Service Wrapper and Annotator Wrapper, see Figure 4(a) and Figure 4(b) respectively. The former is used to wrap an annotator as a language service. The latter is used to wrap a language service as an annotator. CDEF Maker and CDEF Extractor have slightly different roles in the different wrappers as follows.
The Language Service Wrapper consists of three parts: A CDEF Maker, an annotator, and a CDEF Extractor:

- CDEF Maker creates CDEF document from service input. It first maps the service input type to an annotation type and then forms a CDEF document that includes the annotation.

- The Annotator analyses CDEF document and add analytics result to the document.

- CDEF Extractor manipulates with the CDEF document to extract from <*fs*> node and sub-nodes <*f*>*s* the annotations produced by the annotator. Then it maps each annotation to its corresponding language service type as found in the output of a language service.

The Annotator Wrapper is defined as three components: CDEF Extractor, language service, and CDEF Maker:

- The CDEF Extractor manipulates with input CDEF document to extract annotations from <*fs*> node and sub-nodes <*f*>*s*. It then maps the annotation to the

corresponding language service type found in the input of the language service. Position information of the annotations is also extracted.

- The Language service is invoked with the input to produce an output with defined type.
- CDEF Maker maps structure of the language service output to the structure of a corresponding annotation type and use annotation's position information, extracted by CDEF Extractor, to create a CDEF document. This CDEF document becomes output of the annotator.

With this mapping of stand-off annotation and service interface invocation, processing pipeline components can be wrapped as services in service composition frameworks and can be combined with other services to define composite services. Language services can be wrapped as an annotator and inserted into a pipeline flow.

3.2. Integrating Workflow and Pipeline Engines

To realize the integration, we propose pipeline service which uses pipelines to create service workflows. A pipeline service is a composite service created from a pipeline flow. This pipeline is comprised by several annotators, where the annotators can be pipeline approach components or wrapped language services. The pipeline service architecture consists of two components: Service Invoker and Pipeline Executor. When users send requests to a pipeline service, the Service Invoker will start to execute the composite service, it sets binding information for composite service and triggers the Pipeline Executor to execute the pipeline. Pipeline engine is used to run the input information through the pipeline.
With this pipeline service we can use pipeline engine to create and execute composite services. We argue that by declaring a pipeline as a Web service invocation, the integration between Service workflow approach and Processing pipeline approach becomes practical.
Moreover, pipeline services allow the pipeline engine to support the service composition approach for service workflow execution. Pipelined and Parallel execution features of pipeline engine may help to improve performance of composite services. Data parallelism can be used to pass large documents, split into smaller data partitions, through a composite service. These data partitions will be processed by multiple instances of the composite services in multiple parallel threads simultaneously. This may helps to improve the throughput of the workflow.

3.3. Integration Framework

Integration framework enables users easily combine two types of components: language service and annotator. The latter yield composite services and can use language services in pipeline manner. This increases the interoperability of the two types of framework. It also provides the ability to include the pipeline engine in a composite service.
Figure 5 shows that the integration of service composition and processing pipeline uses a wrapper system consisting of Language Service Wrapper and Annotator Wrapper. Annotator providers use the Language Service Wrapper to wrap

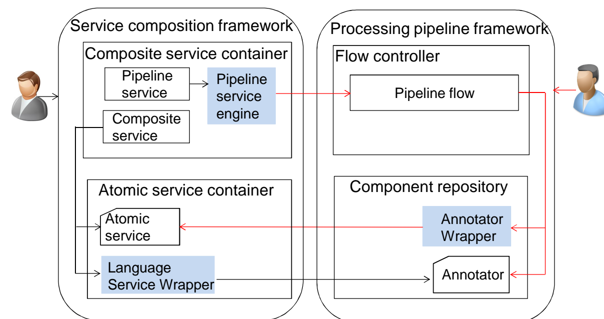


Figure 5: Integration Framework

annotators as language services. A service is then shared with access control in the service composition framework. Users who have access rights to the system can invoke this service or can use this service to compose composite services. The language service providers use the Annotator Wrapper to wrap a language service into an annotator, this annotator can be executed and combined in a pipeline flow. Language resources are shared as language services with protection of intellectual rights making it easy to create composite services. However, pipeline flow has better performance than the comparable composite service when processing large amounts of data. We wrap the pipeline service engine as a service composition engine to create composite services. The pipeline service with parallel execution inherited from pipeline flow will have better performance when processing huge amounts of data.
Moreover, the integration framework helps users easily switch between workflow engine and pipeline engine to execute workflows. Users can also define hybrid workflow which uses both engines to execute the workflow, pipeline engine is used to executed a part of the workflow to help improves thoughput of this part and so increase the performance of the whole workflow.

4. Case study: Integration of the Language Grid and UIMA

In this section we realize the integration framework concept by integrating the Language Grid and UIMA.

4.1. Benefits of Integrating Language Grid and UIMA

Originally launched by IBM, UIMA, the Unstructured Information Management Architecture, is an open-platform middleware for dealing with unstructured information (text, speech, audio, video data). The Apache Software Foundation has established an open-source project for developing UIMA-based software. In addition, the Organization for the Advancement of Structured Information Standards (OASIS) has created a Technical Committee for UIMA standardization. Accordingly, an increasing number of NLP research institutes as well as Human Language Technology companies all over the world are basing their software development efforts on UIMA. Several research institutes and universities have created repositories for UIMA-compliant analytics. For example, Carnegie Mellon University's Lan-

guage Technology Institute has hosted an UIMA Component Repository website [1], where developers can post information about their UIMA analytics. U-Compare (Kano et al., 2011) introduced the world largest repository of ready-to-use type compatible UIMA components. However, currently there is no platform to share UIMA-compliant language resources while protecting intellectual property rights. This limits the accessibility and usability of the language resources. Users who want to use language resources have to find and negotiate with the owners to get these language resources and freely combine them in their applications without considering other stakeholders. Once the language resources are passed to users, the providers will have no control on their language resources.

Unlike UIMA, the purpose of the Language Grid is to support intercultural collaboration by service workflows. A workflow combines language resources and their attendant complex intellectual property issues, such as machine translators, parallel corpora, and bilingual dictionaries. Language resource providers retain ownership of their language resources shared on the Language Grid. Providers can monitor statistical usage information of their language resources, and can also establish access control for the resources. These functions may encourage providers to share language resources on the Language Grid, increasing the availability and accessibility of language resources.

Integration of the Language Grid and UIMA can help NLP communities share their UIMA-compliant language resources via the Language Grid. This increases the number of language services available on the Language Grid. Users can easily find, access and use language resources for their applications. In addition, users can use the UIMA engine to define composite services. The UIMA engine with its data parallelism support can increase the performance of composite services by decreasing the execution time of composite services when processing huge amounts of data.

4.2. Implementing the Integration

Using the integration framework concept, we integrated the Language Grid and UIMA. We implemented two wrappers: Language Service Wrapper and Analysis Engine Wrapper. The former is used to wrap an UIMA analysis engine as a language service, while the latter is used to wrap language service into an Analysis Engine.

UIMA is a data-driven architecture which means that single components communicate with each other by exchanging data. The data exchanged includes the original document and the annotations produced by each component. This data is structured as per the Common Analysis Structure (CAS). CAS is one of the CDEF formats used in the UIMA framework. UIMA specifies the main interface for the analysis engine. This interface takes CAS-input and delivers CAS-output. As one part of UIMA, an XML specification for CAS has been defined. Figure 6 shows a simple example of CAS in XML format. It is possible to develop UIMA analysis engines that directly accept and output this XML. However, to support analysis engines that consume or produce complex data structures, UIMA framework provides a native language interface to the CAS. This has been implemented in both C++ and Java (Ferrucci and Lally, 2004a). Based on this interface we implement CAS Maker and CAS Extractor to manipulate with CAS documents and create the wrappers:

```
<?xml version="1.0" encoding="UTF-8"?>
<xmi:XMI xmlns:cas="http:///uima/cas.ecore"
        xmlns:xmi="http://www.omg.org/XMI"
        xmlns:tokenizer="http:///org/apache/uima/examples/tokenizer.ecore"
        xmlns:tcas="http:///uima/tcas.ecore"
        xmlns:examples="http:///org/apache/uima/examples.ecore" xmi:version="2.0">
    <cas:Sofa xmi:id="1" sofaNum="1" mimeType="text"
            sofaString="This is a simple example."/>
    <tcas:DocumentAnnotation xmi:id="8" sofa="1" begin="0" end="26" language="en"/>
    <examples:SourceDocumentInformation xmi:id="13" sofa="1" begin="0" end="0"
            uri="file:simple example.txt" offsetInSource="0"
            documentSize="28" lastSegment="false"/>
    <tokenizer:Sentence xmi:id="21" sofa="1" begin="0" end="26"/>
    <tokenizer:Token xmi:id="25" sofa="1" begin="0" end="5"/>
    <tokenizer:Token xmi:id="29" sofa="1" begin="6" end="8"/>
    <tokenizer:Token xmi:id="33" sofa="1" begin="9" end="10"/>
    <tokenizer:Token xmi:id="37" sofa="1" begin="11" end="17"/>
    <tokenizer:Token xmi:id="41" sofa="1" begin="18" end="25"/>
    <tokenizer:Token xmi:id="45" sofa="1" begin="25" end="26"/>
    <cas:View sofa="1" members="8 13 21 25 29 33 37 41 45"/>
</xmi:XMI>
```

Figure 6: Example of UIMA CAS

- CAS Extractor extracts annotations from CAS document and maps input/output of language services.
- CAS Maker maps the input/output types of language services with UIMA annotation types and creates CAS documents which serves as input/output of the analysis engine.

In order to create Language Service Wrapper we define a new language service interface to represent an analysis engine. The operations of this interface include analyzing the CAS input and output. Figure 7 shows class diagram of this wrapper. There is new web service interface in the Language Grid node, which is located in the package *jp.go.nict.langrid.wrapper.uima*. Developers can wrap their analysis engines by implementing and extending *UIMAAEService* and *AbstractUIMAAEService*. The implementation of this wrapper architecture is as follows:

- There are three additional created classes: *UIMAComponent* (defines analysis engine to be wrapped into the language service), *UIMAAEService* (defines interface for service access), and *AbstractUIMAAEService* (implements this interface).
- The Concrete Service class (e.g. *MorphologicalService*) defines a concrete UIMA service which extends *AbstracUIMAService* and use UIMAAE to call process method in an analysis engine.

Analysis Engine Wrapper is used to wrap a language service into an analysis engine. Figure 8 shows the class diagram used to implement this wrapper. There are several new classes defined as follows:

- *CASExtractor* is defined to extract annotations from the CAS and map them to the input of the language service.

[1]http://uima.lti.cs.cum.edu

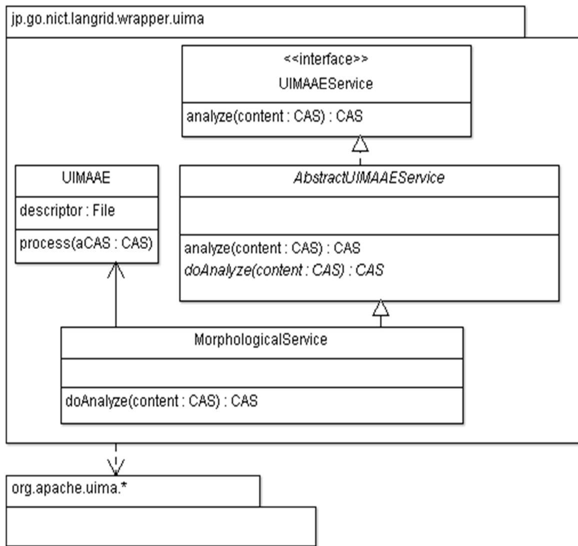


Figure 7: Class diagram of language service wrapper

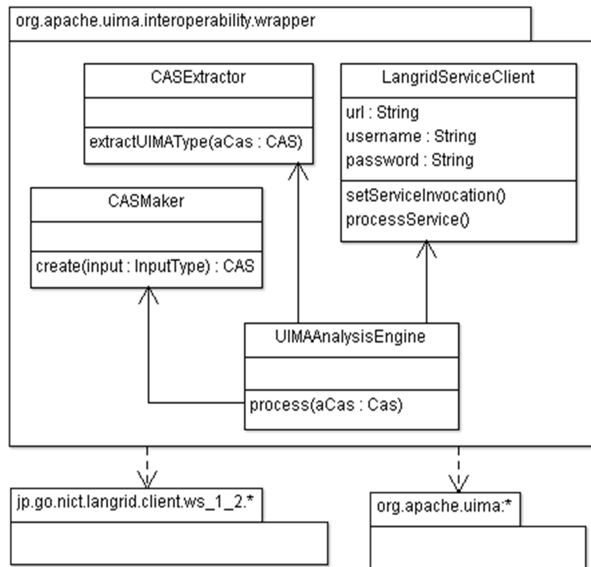


Figure 8: Class diagram of analysis engine wrapper

- *CASMaker* is used to map output of the language service to a corresponding annotation type and add this annotation to the CAS document.
- *LangridServiceClient* is used to set language service endpoint and invoke the service with the input extracted from the CAS.
- *UIMAAnalysisEngine* invokes a language service with the input extracted from CAS document, output of the service is added to the CAS document by using CAS Maker.

We adapt UIMA Engine into the Language Grid Composite Service Container (Murakami et al., 2011), so that users can use this engine to create composite services from UIMA aggregate analysis engines (UIMA AAE). Figure 9 show class diagram used for implementing this composite service. We use UIMA AAEs to define a pipeline of several Analysis Engine Wrappers. In the Analysis Engine Wrappers we use *ComponentServiceFactory.getService* to set service type to be invoked. We can then create composite services by calling the UIMA AAEs. These composite services are shared on composite service container in the Language Grid. When a request is sent to the UIMA composite service, the Java Method Invoker will set invocation information for the composite service, UIMA engine will be used to execute the workflow.

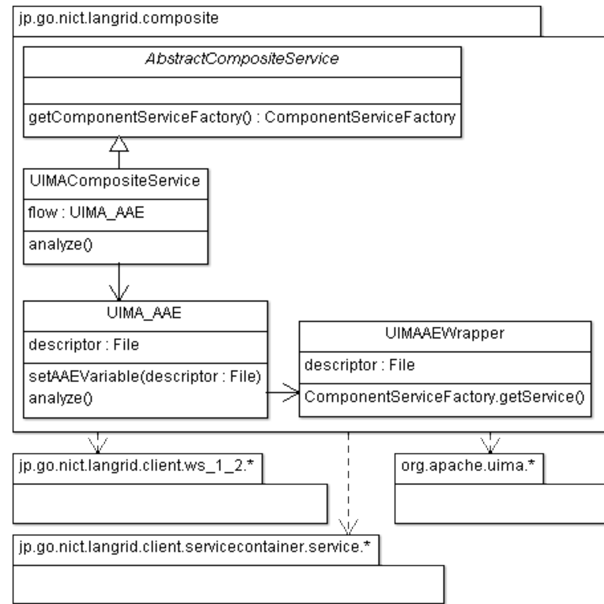


Figure 9: Class diagram of UIMA composite service

UIMA supports data parallelism to improve pipelines performance with CAS Multiplier and UIMA Asynchronous Scale-out (AS) (Epstein et al., 2012). CAS Multiplier separates a big CAS into smaller CASes and put them into a CAS pool. UIMA AS is use to execute multiple CASes with multiple instances of a flow in parallel. Figure 10 shows data parallelism concept in UIMA framework. Users can define number of concurrent instances of the flow according to computing resources of their environments. Using UIMA CAS Multiplier and UIMA AS engine to create and execute composite service will improve composite service performance when processing huge amounts of data. We can use UIMA engine to execute a part of a workflow to improve thoughput of this part and so increase the performance of the workflow. For example, we created a Language Grid hybrid composite service of translation combines with dictionary for Japanese-German specialized translation as show in figure 11. The translation from Japanese to German is a two-hop translation created from a UIMA Aggregate Analysis Engine (UIMA AAE). This UIMA AAE consists of two translation analysis engines wrapped from translation language services. In this example we have realized the integration framework to create a hybrid composite service, we can employ not only workflow engine but also UIMA pipeline engine as Workflow Executor for composite services.

In addition, this proposal allows us to integrate the Language Grid with many UIMA-based frameworks such as U-

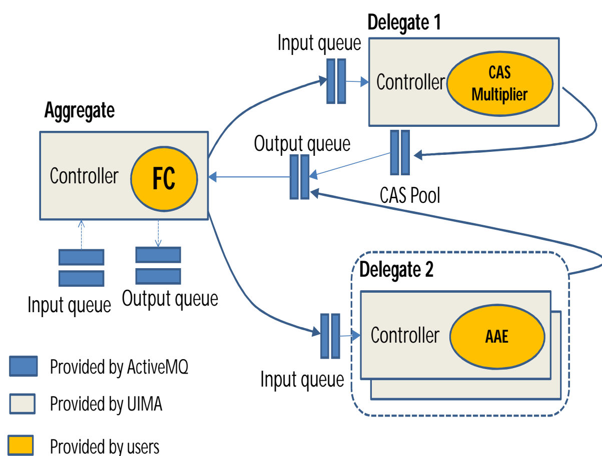


Figure 10: Data Parallelism in UIMA

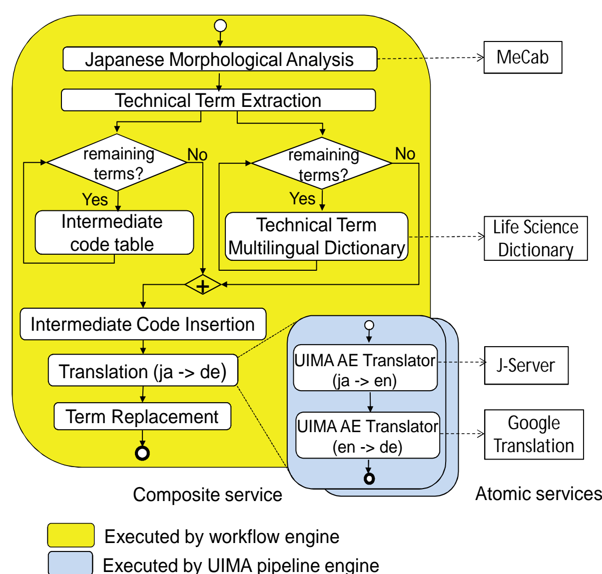


Figure 11: Example of hybrid composite service

Compare (Kano et al., 2011) one of the World's largest, and still growing, collection of UIMA-compatible resources. This would significantly increase the number of NLP-related language resources available as services in the Language Grid. NLP communities will benefits from a large number of easy access language services shared on the Language Grid.

4.3. Evaluation

We performed an evaluation to investigate the performance of composite service when using UIMA CAS Multiplier and UIMA AS. The performance metric is execution time of the composite service when processing a big document. To evaluate composite service performance, we use the previous example in Figure 11, the two-hop translation is implemented with two methods: Using service composition engine in the Language Grid, and using UIMA engine with CAS Multiplier and UIMA AS. To execute the two-hop translation with the latter implementation, we first need to separate large document to smaller documents. We implement a simple Segmenter to separate large document

Figure 12: Execution time of different types of composite service

to smaller CAS documents each consisting of 500 words. Number of smaller CAS documents, which can be executed concurrently, depends on the configuration of the UIMA composite service.

We conducted an experiment to see how composite service performance was improved when using UIMA AS and CAS Multiplier. We assume that the number of composite service instances, which can be executed in the same time, may also affect execution time of the composite service. We designed six UIMA composite services (UIMA AAE) with different parameter specifying number of instances that can be executed concurrently. For example, UIMA AAE 1 can run 10 concurrent 10 CASes simultaneously.

We challenged each type with a data set containing ten text documents with different sizes, ranging from 5000 words to 50000 words. The CAS Multiplier can separate these ten documents into different number of CASes, ranging from 10 CASes to 100 CASes. We compare the execution times of these composite service with execution time of normal two-hop translation service. The resulting execution times are plotted in Figure 12.

From this result we can see that, the CAS Multiplier and UIMA AS engine improve the composite service performance significantly. For example, when we use UIMA AAE 6 to execute with document 10, the execution time was decreased almost 70 % compared to the execution time of normal two-hop translation service.

5. Discussion

The previous section presented an integration of the Language Grid and UIMA. The experiment showed that using data parallelism in a composite service improves its performance. However, if there are too many data partitions, many requests will be sent to a web service concurrently. This is a challenge for service handlers if the number of requests exceeds the maximum number of requests that the service can handle. Therefore, to use data parallelism effectively developers need to design a suitable partition strategy

for the input document. There are other parameters beside the number of concurrent composite service instances that effect performance of the composite services. For example, computing resources and service loads are important parameters that need to be considered to find an optimal setting for a composite service in a given environment.

Trying to optimize the parameters involves several trade-offs. The local computing resource defines the maximum number of threads that can be executed. If the number of concurrent instances exceeds this maximum number, the performance of the composite service would be decreased. If the number of parallel requests exceeds the maximum number of requests that a service can handle, composite service performance might worsen. It will be more interesting in the future when we have more researches in analysing data parallelism in composite service. Further experiment need to be conducted in order to build a correlation model of parameters for analysing and finding optimal parameters for composite service on a given environment.

6. Conclusion

In this paper we proposed an integration of two types of language resource frameworks: framework-based service composition and framework-based processing pipeline. The main contributions of this paper are as follows:

- Adapting a pipeline engine to yield a service composition engine for enhancing composite service performance.
- The integration framework increases the availability of language resources by protecting intellectual property rights.
- A case study that showed a promising increase in the number of language services in the Language Grid after integration with UIMA. This leads to greater use of the Language Grid

Adding the UIMA framework to the Language Grid will in-crease the number of language resources shared as language services on the Language Grid and increase its robustness. It also allows NLP users to easily access and test UIMA-compliant language resources before downloading these language resources and combining them to build their applications.

7. Acknowledgements

This research was partly supported by a Grant-in-Aid for Scientific Research (S) (24220002, 2012-2016) from Japan Society for Promotion of Science (JSPS) and Service Science, Solutions and Foundation Integrated Research Program from JST RISTEX.

8. References

Bel, N. (2010). Platform for automatic, normalized annotation and cost-effective acquisition of language resources for human language technologies. panacea. *Procesamiento del Lenguaje Natural*, 45:327–328.

Cunningham, H., Maynard, D., Bontcheva, K., and Tablan, V. (2002). Gate: an architecture for development of robust human language technology applications. In *Proceedings of 40th Annual Meeting of the Association for Computational Linguistics*, pages 168–175, Philadelphia, Pennsylvania, USA, July. Association for Computational Linguistics.

Epstein, E. A., Schor, M. I., Iyer, B. S., Lally, A., Brown, E. W., and Cwiklik, J. (2012). Making watson fast. *IBM Journal of Research and Development*, 56(3.4):15–1.

Ferrucci, D. and Lally, A. (2004a). Building an example application with the unstructured information management architecture. *IBM Systems Journal*, 43(3):455–475.

Ferrucci, D. and Lally, A. (2004b). Uima: an architectural approach to unstructured information processing in the corporate research environment. *Natural Language Engineering*, 10(3-4):327–348.

Ferrucci, D., Brown, E., Chu-Carroll, J., Fan, J., Gondek, D., Kalyanpur, A. A., Lally, A., Murdock, J. W., Nyberg, E., Prager, J., et al. (2010). Building watson: An overview of the deepqa project. *AI magazine*, 31(3):59–79.

Hayashi, Y., Declerck, T., Calzolari, N., Monachini, M., Soria, C., and Buitelaar, P. (2011). Language service ontology. In *The Language Grid*, pages 85–100. Springer.

Ide, N. and Romary, L. (2009). Standards for language resources. *arXiv preprint arXiv:0909.2719*.

Ide, N. and Suderman, K. (2009). Bridging the gaps: interoperability for graf, gate, and uima. In *Proceedings of the Third Linguistic Annotation Workshop*, pages 27–34. Association for Computational Linguistics.

Ide, N., Bonhomme, P., and Romary, L. (2000). An xml-based encoding standard for linguistic corpora. In *Proceedings of the Second International Conference on Language Resources and Evaluation*, pages 825–830.

Ishida, T. (2006). Language grid: An infrastructure for intercultural collaboration. In *Applications and the Internet, 2006. SAINT 2006. International Symposium on*, pages 96–100. IEEE.

Kano, Y., Miwa, M., Cohen, K. B., Hunter, L. E., Ananiadou, S., and Tsujii, J. (2011). U-compare: A modular nlp workflow construction and evaluation system. *IBM Journal of Research and Development*, 55(3):11–1.

Maegaard, B., Choukri, K., Calzolari, N., and Odijk, J. (2005). Elra - european language resources association-background, recent developments and future perspectives. *Language Resources and Evaluation*, 39(1):9–23.

Murakami, Y., Lin, D., Tanaka, M., Nakaguchi, T., and Ishida, T. (2011). Service grid architecture. In *The Language Grid*, pages 19–34. Springer.

Vanhoutte, E. (2004). An introduction to the tei and the tei consortium. *Literary and linguistic computing*, 19(1):9–16.

Zhang, D., Coddington, P., and Wendelborn, A. L. (2013). Improving data transfer performance of web service workflows in the cloud environment. *Int. J. Comput. Sci. Eng.*, 8(3):198–209, July.