An Ontology for Language Service Composability

Yohei Murakami
Unit of Design,
Kyoto University
yohei@i.kyoto-u.ac.jp

Takao Nakaguchi
Department of Social Informatics
Kyoto University
nakaguchi@i.kyoto-u.ac.jp

Donghui Lin
Department of Social Informatics
Kyoto University
lindh@i.kyoto-u.ac.jp

Toru Ishida
Department of Social Informatics
Kyoto University
ishida@i.kyoto-u.ac.jp

Abstract

Fragmentation and *recombination* is a key to create customized language environments for supporting various intercultural activities. *Fragmentation* provides various language resource components for the customized language environments and *recombination* builds each language environment according to user's request by combining these components. To realize this fragmentation and recombination process, existing language resources (both data and programs) should be shared as language services and combined beyond mismatch of their service interfaces. To address this issue, standardization is inevitable: standardized interfaces are necessary for language services as well as data format required for language resources. Therefore, we have constructed a hierarchy of language services based on inheritance of service interfaces, which is called *language service ontology*. This ontology allows users to create a new customized language service that is compatible with existing ones. Moreover, we have developed a dynamic service binding technology that instantiates various executable customized services from an abstract workflow according to user's request. By using the ontology and service binding together, users can bind the instantiated language service to another abstract workflow for a new customized one.

1 Introduction

Rapid internationalization accelerates expansion of multicultural society where local people and foreigners coexist. As a result, intercultural and multilingual activities are often necessary in daily life, such as questioning foreign patients in hospitals and teaching foreign students in schools, and so on. Although there are many language resources (e.g. bilingual dictionaries, parallel corpora, machine translators, morphological analyzers, and so on) on the Internet(Choukri, 2004), most intercultural collaboration activities are still lacking multilingual support. This is because each activity requires a customized language environment due to different purposes, domains, environments, and languages to be supported.

Many efforts have been put for combining language resources in some previous frameworks based on web services. These efforts focus on wrapping the resources as services, and defining a standard data format exchanged between language services and annotation vocabularies to be embedded in the format. The format and vocabularies enable users to combine language services developed by different providers. However, end users who need multilingual support in their intercultural fields have difficulties in developing a reasonable logic flow to combine language services because they are not familiar with the annotation vocabularies. Therefore, we aim at separating the logic flow from selecting language services by introducing an abstract workflow into our platform. In our platform, web service professionals develop an abstract workflow to combine language services, while users select language services to be connected by the workflow in order to instantiate their customized language services from the abstract workflow.

To realize this collaboration between web service professionals and end users, we addressed the following two issues.

Construct an ontology to define language services As web service professionals develop new language services according to users' request, it results in generating various service interfaces. To invoke the new services from existing workflows while ensuring the diversity, we need an ontology to verify composability of the new services.

Develop language service binding technology To instantiate various executable composite language services from an abstract workflow, we need a technology that enables users to bind any language services to the workflow when executing the workflow.

The remainder of this paper is organized as follows: in section 2 we will briefly discuss features of the two approaches to combine language services. And then, our framework called *Service Grid* and our proposed *language service ontology* will be presented in section 3 and 4. Moreover, we show how to bridge our language services with ones in different frameworks in section 5. Finally, section 6 concludes this paper.

2 Related Works

There are two types of frameworks supporting developers to combine language resources: pipeline processing approach such as GATE(Cunningham et al., 2002) and UIMA(Ferrucci and Lally, 2004) (U-compare(Kano et al., 2009) and DKPro(Eckart de Castilho and Gurevych, 2014)), and service composition approach such as PANACEA(Toral et al., 2011), LAPPS Grid(Ide et al., 2014), and the Language Grid (Ishida, 2011). The former focuses on processing a huge amount of data at local with pipeline technique. On the other hand, the latter aims to share language resources distributed on the Internet as a web service and combine them in a workflow manner. As shown in Table 1, we summarized features of these frameworks from view point of interfaces, data format, and type system.

2.1 Pipeline Processing Approach

This approach employs a common interfaces and common data format exchanged between language resources. The resources are combined into a pipeline to analyze documents. Each resource called a processing resource or analysis engine annotates a document flowed in the pipeline in the stand-off annotation manner. The document together with annotations is formed in a common data format, such as GATE format and CAS (Common Analysis Structure). The annotations comply with pre-defined annotation type system. GATE provides annotation schemas to define annotation type for each case, while U-Compare and DKPro contain pre-defined annotation types based on UIMA type system.

2.2 Service Composition Approach

This approach wraps language resources as web services, called *language services*, and standardizes the language service interfaces. LAPPS Grid provides a single common method same as the pipeline processing approach, while PANACEA and Language Grid define the interfaces according to language service types because the workflow can freely assign outputs of a service to inputs of another one. The unique feature of this approach is to combine language services distributed in the Internet. This feature requires interoperability of language services on different frameworks. To satisfy this requirement, LAPPS Grid and PANACEA present several format converters between different format and their own formats like LIF (LAPPS Interchange Format)(Verhagen et al., 2016) and Travelling Object. Moreover, LAPPS Grid defines LAPPS Web Service Exchange Vocabulary(Ide et al., 2016), an ontology of terms for a core of linguistic objects and features exchanged among language services that consume and produce linguistically annotated data. It is used for service description and input/output interchange to support service discovery and composition. On the other hand, Language Grid enforces service providers to wrap their resources with the standardized interfaces that expose every annotation data type, in order

Table 1: Comparison among Frameworks

	GATE	U-Compare	DKPro	LAPPS Grid	PANACEA	Language Grid
Standardized Interface	✓ (common)	✓ (common)	✓ (common)	✓ (common)	✓ (each type)	✓ (each type)
Common Format	✓ (GATE format)	✓ (CAS)	✓ (CAS)	✓ (LIF)	✓ (Traveling Object)	
Format Converter				✓	✓	
Type System (Vocabularies)	✓	✓	✓	✓		✓

to remove the needs of a common format and converters. To bridge other service-based frameworks and Language Grid, we have developed adapters that adapt Heart of Gold and UIMA component to our standardized interfaces(Bramantoro et al., 2008; Trang et al., 2014).

The existing frameworks except for the Language Grid combine instances of language services in a pipeline and workflow. The pipeline and workflow that tightly couple language services need to be modified when changing a language service according to user's request. This becomes a barrier for end users to create their customized services by themselves. Therefore, we have introduced an abstract workflow that is composed of interfaces of components and their dependencies. The abstract workflow separates binding language services from designing it so that users can reuse the workflow to instantiate it with combination of language services they want. In this composition method that delays service binding, it is significant to verify which language services are compatible with the designed workflows. Therefore, we describe two ontologies that defines language service type and organizes a hierarchy of language services in section 3 and 4.

3 Service Grid

Interoperability of language services requires standardization of service interfaces and metadata according to their functionalities. To this end, our service grid provides a service grid ontology for operators to organize services in their domain into several service classes (Murakami et al., 2012). As illustrated in Figure 1, the service grid ontology is not just an ontology of data exchanged between services, but an ontology to define service metadata and resource metadata. ServiceGrid class has more than one Resource class and Service class that is provided from the corresponding Resource class. Resource and Service class have more than one attribute to describe features of their instances. Also, Service class has one service interface to allow users to access the service instances via several protocols. A service grid operator can define his domain service grid ontology by inheriting ServiceGrid, Resource, and Service class.

Based on the service grid ontology, we constructed Language Grid Ontology shown in Figure 2. This

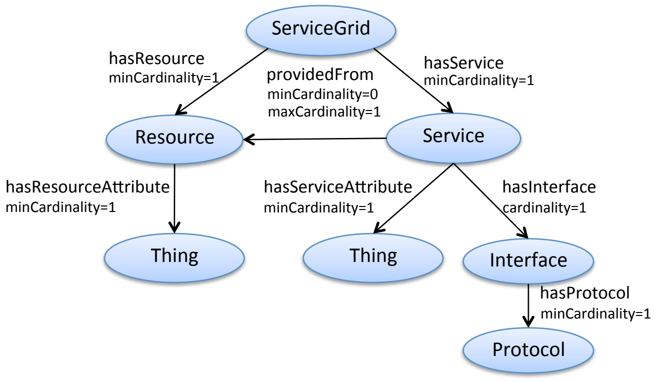

Figure 1: Service Grid Ontology

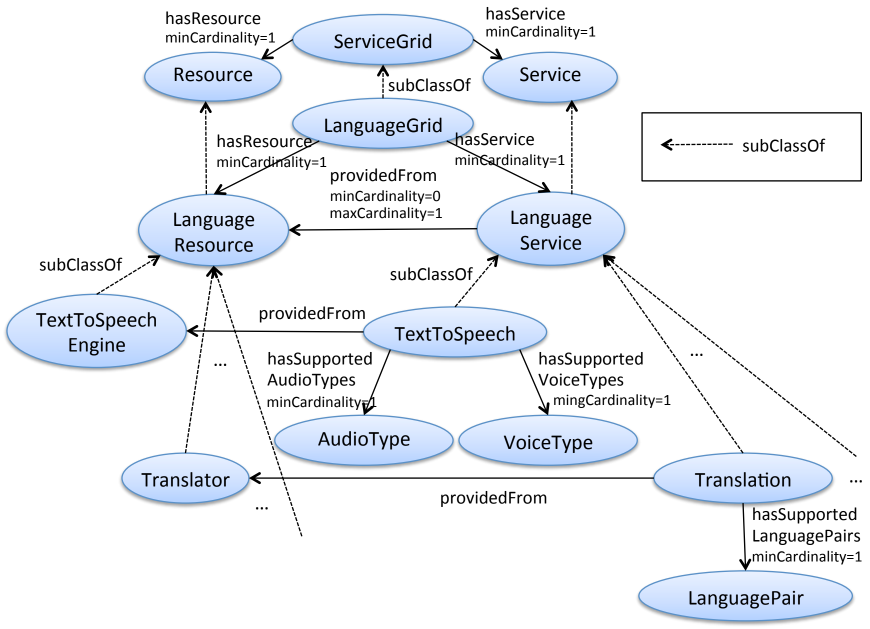

Figure 2: Inheriting Service Grid Ontology

ontology defines LanguageGrid, LanguageResource, and LanguageService classes as subclasses of ServiceGrid, Resource, and Service classes. Moreover, the LanguageResource and LanguageService classes derive 14 types of language resource classes and 17 types of language service classes such as text to speech engines, translator, and so on. Table 2 shows the language service classes. These service classes are characterized with hasServiceAttribute property, indicating which objects a given service can process and which methods the service can employ. The former is *hasSupportedLanguages*, *hasSupportedLanguagePairs*, *hasSupportedLanguagePaths*, *hasSupportedImageTypes*, *hasSupportedAudioTypes*, and *hasSupportedVoiceTypes*. They are used to specify languages, images, and audio files to be processed by services. The latter is *hasSupportedMatchingMethod*. This is used to specify search functionalities implemented on language data such as bilingual dictionaries, concept dictionaries, and so on.

Moreover, we defined a service interface for each service class. To standardize the interface, we extracted common parameters of language resources belonging to the same resource type. In case of morphological analyzers, we have several morphological analysis services according to supported languages: TreeTagger, MeCab, Juman, KLT, ICTCLAS. A source text and source language for input parameters are common among all the existing morphological analyzers. On the other hand, we have many formats of morphemes for output parameters. Every analyzer returns word, lemma, and part of speech tag except for ICTCLAS. Therefore, we defined the output of morphological analysis service as an array of triples consisting of word, lemma, and POS tag. Furthermore, we enumerated POS tags available in the output of the analysis service. Since POS tags vary depending on languages, we selected a minimal set of POS tags occurring in every language: noun, proper noun, pronoun, verb, adjective, adverb, unknown, and other. Most morphological analyzers can be wrapped with this standard interface. A few morphological analyzers not complying with this interface, such as ICTCLAS, return ”NULL” as unassigned parameters. This interface is designed for interoperability instead of completeness. As a result, information generated by the original morphological analyzers can be lost.

Due to limitations of space, Table 2 shows only the operation name of Interface class except for input and output parameters. Refer to `http://langrid.org/service_manager/service-type` for the WSDL files and more information. The attributes and interfaces help service users to compose services by searching services with the metadata and changing the services belonging to the same service type.

Table 2: Language Service Classes

Service class	hasServiceAttribute property	Thing class	Interface class
BackTranslation	hasSupportedLanguagePaths	LanguagePath	backtranslate
BilingualDictionary	hasSupportedLanguagePairs, hasSupportedMatchingMethods	LanguagePair, MatchingMethod	search
BilingualDictionaryWith LongestMatchSearch	hasSupportedLanguagePairs, hasSupportedMatchingMethods	LanguagePair, MatchingMethod	search searchWithLongestMatch
ConceptDictionary	hasSupportedLanguages, hasSupportedMatchingMethods	Language, MatchingMethod	searchConcepts, getRelatedConcepts
DependencyParse	hasSupportedLanguages	Language	parseDependency
DialogCorpus	hasSupportedLanguages, hasSupportedMatchingMethods	Language, MatchingMethod	search
LanguageIdentification	hasSupportedEncodings, hasSupportedLanguages	Encoding, Language	identify
MorphologicalAnalysis	hasSupportedLanguages	Language	analyze
MultihopTranslation	hasSupportedLanguagePaths	LanguagePath	translate multihopTranslate
NamedEntityTagging	hasSupportedLanguages	Language	tag
ParallelText	hasSupportedLanguagePairs, hasSupportedMatchingMethods	LanguagePair, MatchingMethod	search
Paraphrase	hasSupportedLanguages	Language	paraphrase
PictogramDictionary	hasSupportedLanguages, hasSupportedMatchingMethods, hasSupportedImageTypes	Language, MatchingMethod, ImageType	search
SimilarityCalculation	hasSupportedLanguages	Language	calculate
SpeechRecognition	hasSupportedLanguages, hasSupportedAudioTypes, hasSupportedVoiceTypes	Language, AudioType, VoiceType	recognize
TextToSpeech	hasSupportedLanguages, hasSupportedAudioTypes, hasSupportedVoiceTypes	Language, AudioType, VoiceType	speak
Translation	hasSupportedLanguagePairs	LanguagePair	translate
TranslationWith TemporalDictionary	hasSupportedLanguagePairs	LanguagePair	translate translateWithDict

4 Language Service Ontology

Service Grid ontology allows operators to add a new service type according to users' requests. As the number of service types increases, the reusability of workflows decreases because the service grid may have many close but not the same interfaces for the common functionalities. For example, when many service users need more detailed information of morphological analysis services, a new one may be added. To increase the reusability of workflows, it is significant to verify which language services are compatible with the existing workflows. In this section, we firstly describe semantic matching that guarantees substitutability of language services in an abstract workflow. Based on the semantic matching, we then construct a hierarchy of language services, and explain how to bind language services to the workflow.

4.1 Semantic Matching

Semantic matching was introduced to discover a service whose capability satisfies user's request(Paolucci et al., 2002). Since the goal of the previous research is to fulfill users' request as much as possible but not partially, the semantic matching prefers services that output a superclass of users' required class. However, our goal is to find services whose capabilities are compatible with the existing workflow. If a service that outputs a superclass of user's required class is selected, the subsequent service in a workflow may fail to run because the service has possibilities to receive an input different from its expected input, such as a sibling class. This semantic matching causes a type-unsafety issue. Therefore, we modify the semantic matching rules by considering type-safety in binding services to a workflow.

Firstly, we define notations relevant to an input-output of a service, and then modify the semantic matching rules.

Definition 4.1 (Input-output of a service) *A service s is defined as a tuple $s = \{Input_s, Output_s\} \in S$ where $Input_s$ is a set of inputs required to invoke s, $Output_s$ is the set of outputs returned by s after its execution, and S is the set of all services registered in a service grid. Each input and output is also a*

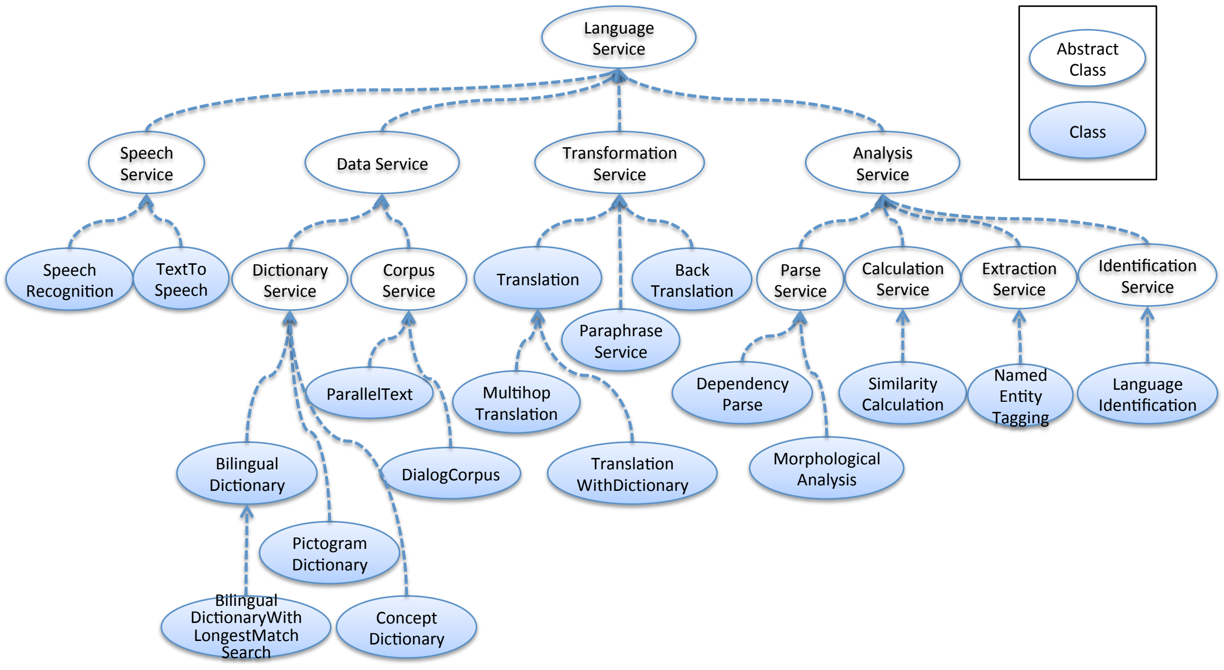

Figure 3: Language Service Ontology

class.

Exact An input $i_s \in Input_s$ and an output $o_s \in Output_s$ of a service s in a set of services S matches an input $i_w \in Input_w$ and output $o_w \in Output_w$ of a service w in a workflow with a degree of exact match if both of input and output classes are equivalent ($i_s \equiv i_w, o_s \equiv o_w$).

Plug-in An input $i_s \in Input_s$ and an output $o_s \in Output_s$ of a service s in a set of services S matches an input $i_w \in Input_w$ and an output$o_w \in Output_w$ of a service w in a workflow with a degree of plugin if i_s is a superclass of i_w ($i_s \sqsupseteq i_w$) and o_s is a subclass of o_w ($o_s \sqsubseteq o_w$).

Subsume An input $i_s \in Input_s$ and an output $o_s \in Output_s$ of a service s in a set of services S matches an input $i_w \in Input_w$ and an output $o_w \in Output_w$ of a service w in a workflow with a degree of subsume if either i_s is a subclass of i_w ($i_s \sqsubset i_w$) or o_s is a superclass of o_w ($o_s \sqsupset o_w$).

Fail When none of the previous matches are found, then both concepts are incompatible and the match has a degree of fail.

Note that, in order to discover type safe services to satisfy data flow in a workflow execution, the only two valid degrees of match are *exact* and *plug-in*. Based on the *exact* match, the Language Grid has introduced inheritance of service interfaces, which guarantees that an inherited interface provides the same methods as the superclass of a service. We construct a hierarchy of language services using the inheritance of service interfaces in 4.2

4.2 Hierarchy of Language Services

When many service users need more additional detailed information, a new service class can be derived by inheriting the service interface of the superclass, but not created from scratch. The inherited service interface can add other interfaces while maintaining the consistency with the existing one. This inheritance of service interfaces constructs a hierarchy of homogeneous services. This is similar to a OWL-S profile hierarchy(Martin et al., 2007) and WSMO capability(Wang et al., 2012). However, they are used to discover alternative services from property aspect but not interfaces like existing taxonomies of service categories.

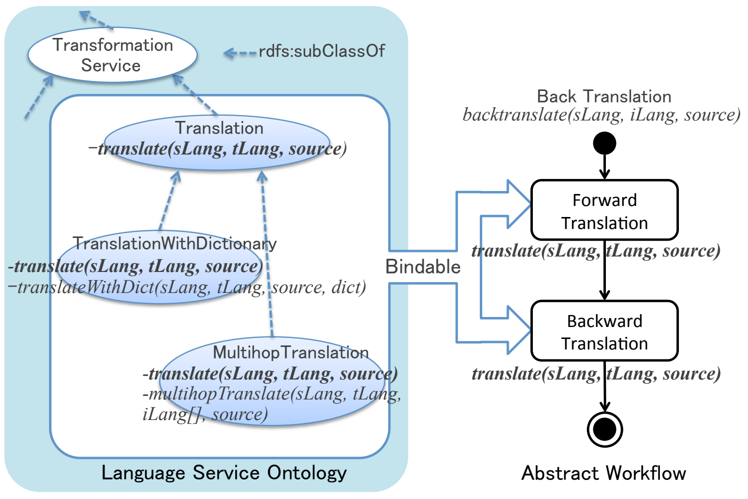
Transformation Service
rdfs:subClassOf
Translation
−translate(sLang, tLang, source)
TranslationWithDictionary
-translate(sLang, tLang, source)
−translateWithDict(sLang, tLang, source, dict)
MultihopTranslation
-translate(sLang, tLang, source)
-multihopTranslate(sLang, tLang, iLang[], source)
Language Service Ontology
Bindable
Back Translation
backtranslate(sLang, iLang, source)
Forward Translation
translate(sLang, tLang, source)
Backward Translation
translate(sLang, tLang, source)
Abstract Workflow

Figure 4: Service Binding with Language Service Ontology

Figure 3 illustrates our hierarchy of language services based on interface inheritance, called *Language Service Ontology*. This ontology consists of abstract classes that have no interface and instance of services and classes that have interfaces and instances. Firstly, LanguageService class is classified into four abstract classes: SpeechService class that processes speech data, DataService class that deals with linguistic data resources like lexical and corpus data, TransformationService class that transforms input texts, and AnalysisService class that analyzes input data like parsing, calculating, extracting, and identifying. By inheriting the abstract classes, we define 18 language service classes listed in Table 2.

Moreover, MultihopTranslation and TranslationWithDictionary class, and BilingualDictionaryWithLongestMatchSearch class are derived from Translation class and BilingualDictionary class, respectively. Hence, they have the same interface as their superclass. For example, as shown in the left side of Figure 4, Translation class provides a translate method whose input parameters are source language, target language, and source text (denoted by sLang, tLang, source, respectively). By extending this interface, TranslationWithDictionary and MultihopTranslation class are defined as a subclass of Translation class. The former introduces simple dictionary data into its input parameters in order to replace words in the translated text with translated words in the dictionary. This aims at improving translation quality by restricting context within the dictionary. Without the dictionary, this service returns a translated text like Translation class. On the other hand, the latter introduces an array of intermediate languages to cascade several translation services. Without intermediate languages, this service behaves like Translation class by using default languages as intermediate languages.

Based on this ontology, we can bind any subclasses to an abstract workflow including the superclass's interface. The right side of Figure 4 shows the case of backtranslation. This workflow connects two translation interfaces to translate the translated text into the source language again. Therefore, we can select JServer, an instance of Translation class, and DictTrans, an instance of TranslationWithDictionary, as a forward translation and backward translation, respectively. To dynamically bind these services in invoking the backtranslation service, we have introduced a hierarchical service composition description using higher-order functions as below(Nakaguchi et al., 2016). This syntax allows users to nest binding in order to invoke a workflow from another one.

```
syntax :== service "." method "(" (arg ("," arg)* )? ")"
service :== serviceId | serviceBinding
method :== symbol
serviceId :== symbol
serviceBinding :== "bind(" serviceId bindingInfo+ ")"
bindingInfo :== "," invocationId ":" service
invocationId :== symbol
arg :== "'" symbol "'"
symbol :== LETTER+
```

Using this language, we can describe the above service binding as below. After executing this language, this description is translated into a SOAP request embedding the binding information into the header.

```
bind(BackTranslation,
 ForwardTranslation:JServer,
 BackwardTranslation:bind(DictTrans,
  MorphologicalAnalysis:TreeTagger,
  BilingualDictionary:AgriDict,
  Translation:GoogleTranslate
 ).backtranslate('en','ja','Land preparation needs puddling and levee painting.')
```

5 Discussion

Our language service ontology focuses on interoperability among homogeneous language services. Meanwhile, it is also significant to enhance interoperability among heterogenous language services. We have two approaches.

One is to construct central ontology as a hub, which was proposed by (Hayashi et al., 2008). The ontology consists of a top-level ontology and sub-ontologies. The top-level ontology defines the relations among language service class, language processing resource class, language data resource class, and linguistic object class. A language service is provided by an instance of the language processing resource class, whose input and output are instances of linguistic object class. A language data resource consists of instances of the linguistic object class. On the other hand, each sub-ontology organizes classes for language processing resources, language data resources, and linguistic annotations of linguistic objects, respectively.

The other is to connect type systems in different frameworks each other. By mapping LAPPS exchange vocabulary type hierarchy and type system in DKPro and U-Compare with input and output classes of language service interfaces in our ontology, it would be possible to discover type safe services with *plug-in* match as well as *exact* match. Logically, covariant return type of services, which return a subclass of the output class, and contravariant argument type of services, which receive a superclass of the input class can be bindable to an abstract workflow.

6 Conclusions

We have introduced an abstract workflow to separate designing a logic flow and selecting language services. By this abstract workflow, we aim at realizing collaboration between web service professionals who develop a workflow and end users who select their needed language services. To this end, we have constructed language service ontology by inheriting service interfaces to verify composability of language services. Moreover, we have applied higher-order function to develop hierarchical language service binding. By using the ontology and binding technology, end users can instantiate an abstract workflow with type-safe language services.

Acknowledgements

This work was supported by Grant-in-Aid for Scientific Research (S) (24220002) of Japan Society for the Promotion of Science (JSPS).

References

Arif Bramantoro, Masahiro Tanaka, Yohei Murakami, Ulrich Schäfer, and Toru Ishida. 2008. A Hybrid Integrated Architecture for Language Service Composition. In *Proc. of the Sixth International Conference on Web Services (ICWS'08)*, pages 345–352.

Khalid Choukri. 2004. European Language Resources Association History and Recent Developments. In *SCALLA Working Conference KC 14/20.*

Hamish Cunningham, Diana Maynard, Kalina Bontcheva, and Valentin Tablan. 2002. GATE: An Architecture for Development of Robust HLT Applications. In *Proc. of the Fortieth Annual Meeting on Association for Computational Linguistics (ACL'02)*, pages 168–175.

Richard Eckart de Castilho and Iryna Gurevych. 2014. A Broad-Coverage Collection of Portable NLP Components for Building Shareable Analysis Pipelines. In *Proceedings of the Workshop on Open Infrastructures and Analysis Frameworks for HLT*, pages 1–11.

David Ferrucci and Adam Lally. 2004. UIMA: An Architectural Approach to Unstructured Information Processing in the Corporate Research Environment. *Journal of Natural Language Engineering*, 10:327–348.

Yoshihiko Hayashi, Thierry Declerck, Paul Buitelaar, and Monica Monachini. 2008. Ontologies for a Global Language Infrastructure. In *Proc. of the First International Conference on Global Interoperability for Language Resources (ICGL'08)*, pages 105–112.

Nancy Ide, James Pustejovsky, Christopher Cieri, Eric Nyberg, Di Wang, Keith Suderman, Marc Verhagen, and Jonathan Wright. 2014. The Language Application Grid. In *Proceedings of the Ninth International Conference on Language Resources and Evaluation (LREC'14)*, pages 22–30.

Nancy Ide, Keith Suderman, Marc Verhagen, and James Pustejovsky. 2016. The Language Application Grid Web Service Exchange Vocabulary. In *Worldwide Language Service Infrastructure*, pages 18–32. Springer International Publishing.

Toru Ishida, editor. 2011. *The Language Grid: Service-Oriented Collective Intelligence for Language Resource Interoperability*. Springer-Verlag.

Yoshinobu Kano, William Baumgartner, Luke McCrohon, Sophia Ananiadou, Kevin Cohen, Larry Hunter, and Jun'ichi Tsujii. 2009. U-Compare: Share and Compare Text Mining Tools with UIMA. *Bioinformatics*, 25(15):1997–1998.

David Martin, Mark Burstein, Drew McDermott, Sheila McIlraith, Massimo Paolucci, Katia Sycara, Deborah L. McGuinness, Evren Sirin, and Naveen Srinivasan. 2007. Bringing Semantics to Web Services with OWL-S. *World Wide Web*, 10(3):243–277.

Yohei Murakami, Masahiro Tanaka, Donghui Lin, and Toru Ishida. 2012. Service Grid Federation Architecture for Heterogeneous Domains. In *Proc. of the IEEE International Conference on Services Computing (SCC-12)*, pages 539–546.

Takao Nakaguchi, Yohei Murakami, Donghui Lin, and Toru Ishida. 2016. Higher-Order Functrions for Modeling Hierarchical Service Bindings. In *Proc. of the Twelfth International Conference on Web Services (ICWS'08)*, pages 798–803.

Massimo Paolucci, Takahiro Kawamura, Terry R. Payne, and Katia Sycara. 2002. Semantic Matching of Web Services Capabilities. In *Proc. of the First International Semantic Web Conference (ISWC'02)*, pages 333–347.

Antonio Toral, Pavel Pecina, Andy Way, and Marc Poch. 2011. Towards a User-Friendly Webservice Architecture for Statistical Machine Translation in the PANACEA Project. In *Proc. of the 15th Conference of the European Association for Machine Translation (EAMT'11)*, pages 63–70.

Mai Xuan Trang, Yohei Murakami, Donghui Lin, and Toru Ishida. 2014. Integration of Workflow and Pipeline for Language Service Composition. In *Proc. of the 9th International Conference on Language Resources and Evaluation Conference (LREC'14)*, pages 3829–3836.

Marc Verhagen, Keith Suderman, Di Wang, Nancy Ide, Chunqi Shi, Jonathan Wright, and James Pustejovsky. 2016. The LAPPS Interchange Format. In *Worldwide Language Service Infrastructure*, pages 33–47. Springer International Publishing.

Hai H. Wang, Nick Gibbins, Terry R. Payne, and Domenico Redavid. 2012. A Formal Model of the Semantic Web Service Ontology (WSMO). *Information Systems*, 37(1):33–60.