# Making UIMA Truly Interoperable with SPARQL

**Rafal Rak and Sophia Ananiadou**
National Centre for Text Mining
School of Computer Science, University of Manchester
{rafal.rak,sophia.ananiadou}@manchester.ac.uk

## Abstract

Unstructured Information Management Architecture (UIMA) has been gaining popularity in annotating text corpora. The architecture defines common data structures and interfaces to support interoperability of individual processing components working together in a UIMA application. The components exchange data by sharing common type systems—schemata of data type structures—which extend a generic, top-level type system built into UIMA. This flexibility in extending type systems has resulted in the development of repositories of components that share one or several type systems; however, components coming from different repositories, and thus not sharing type systems, remain incompatible. Commonly, this problem has been solved programmatically by implementing UIMA components that perform the alignment of two type systems, an arduous task that is impractical with a growing number of type systems. We alleviate this problem by introducing a conversion mechanism based on SPARQL, a query language for the data retrieval and manipulation of RDF graphs. We provide a UIMA component that serialises data coming from a source component into RDF, executes a user-defined, type-conversion query, and deserialises the updated graph into a target component. The proposed solution encourages ad hoc conversions, enables the usage of heterogeneous components, and facilitates highly customised UIMA applications.

## 1 Introduction

Unstructured Information Management Architecture (UIMA) (Ferrucci and Lally, 2004) is a framework that supports the interoperability of media-processing software components by defining common data structures and interfaces the components exchange and implement. The architecture has been gaining interest from academia and industry alike for the past decade, which resulted in a multitude of UIMA-supporting repositories of analytics. Notable examples include METANET4U components (Thompson et al., 2011) featured in U-Compare[1], DKPro (Gurevych et al., 2007), cTAKES (Savova et al., 2010), BioNLP-UIMA Component Repository (Baumgartner et al., 2008), and JULIE Lab's UIMA Component Repository (JCoRe) (Hahn et al., 2008).

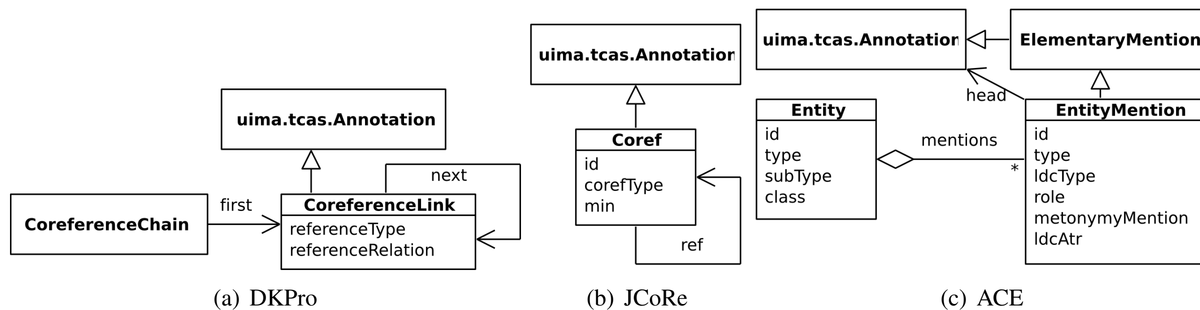
However, despite conforming to the UIMA standard, each repository of analytics usually comes with its own set of *type systems*, i.e., representations of data models that are meant to be shared between analytics and thus ensuring their interoperability. At present, UIMA does not facilitate the alignment of (all or selected) types between type systems, which makes it impossible to combine analytics coming from different repositories without an additional programming effort. For instance, NLP developers may want to use a sentence detector from one repository and a tokeniser from another repository only to learn that the required input `Sentence` type for the tokeniser is defined in a different type system and namespace than the output `Sentence` type of the sentence detector. Although both `Sentence` types represent the same concept and may even have the same set of features (attributes), they are viewed as two distinct types by UIMA.

Less trivial incompatibility arises from the same concept being encoded as structurally different types in different type systems. Figures 1 and 2 show fragments of some of existing type systems;

[1] http://nactem.ac.uk/ucompare/



*Proceedings of the 7th Linguistic Annotation Workshop & Interoperability with Discourse*, pages 89–97,
Sofia, Bulgaria, August 8-9, 2013. ©2013 Association for Computational Linguistics




(a) DKPro (b) JCoRe (c) ACE

Figure 1: UML diagrams representing fragments of type systems that show differences in encoding coreferences.

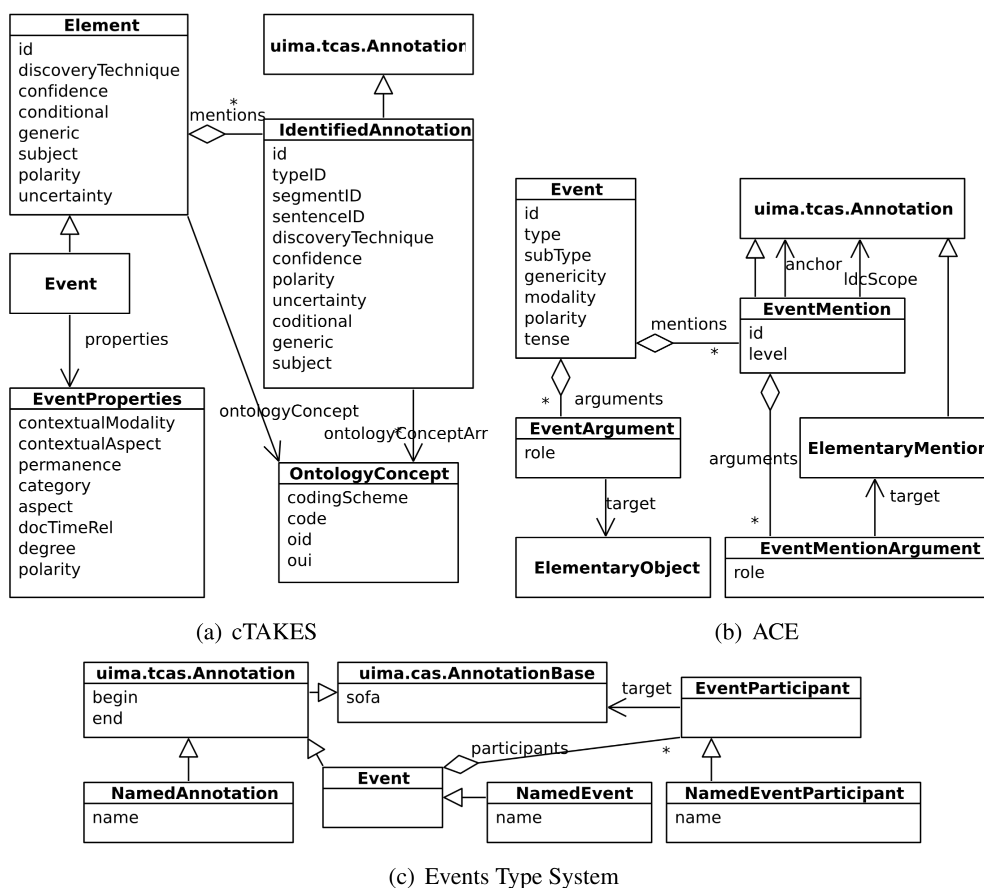
specifically, they show the differences in encoding *coreferences* and *events*, respectively. For instance, in comparison to the JCoRe type system in Figure 1(b), the DKPro type system in Figure 1(a) has an additional type that points to the beginning of the linked list of coreferences.

Conceptually similar types in two different type systems may also be incompatible in terms of the amount of information they convey. Compare, for instance, type systems in Figure 2 that encode a similar concept, event. Not only are they structurally different, but the cTAKES type system in Figure 2(a) also involves a larger number of features than the other two type systems. Although, in this case, the alignment of any two structures cannot be carried out without a loss or deficiency of information, it may still be beneficial to do so for applications that consist of components that either fulfill partially complete information or do not require it altogether.

The available type systems vary greatly in size, their modularity, and intended applicability. The DKPro UIMA software collection, for instance, includes multiple, small-size type systems organised around specific syntactic and semantic concepts, such as part of speech, chunks, and named entities. In contrast, the U-Compare project as well as cTAKES are oriented towards having a single type system. Respectively, the type systems define nearly 300 and 100 syntactic and semantic types, with U-Compare's semantic types biased towards biology and chemistry and cTAKES's covering clinical domain. Most of the U-Compare types extend a fairly expressive higher-level type, which makes them universally applicable, but at the same time, breaks their semantic cohesion. The lack of modularity and the all-embroiling types suggest that the U-Compare type system is developed primarily to work with the U-Compare application.

The Center for Computational Pharmacology (CCP) type system (Verspoor et al., 2009) is a radically different approach to the previous systems. It defines a closed set of top-level types that facilitate the use of external resources, such as databases and ontologies. This gives the advantage of having a nonvolatile type system, indifferent to changes in the external resources, as well as greater flexibility in handling some semantic models that would otherwise be impossible to encode in a UIMA type system. On the other hand, such an approach shifts the handling of interoperability from UIMA to applications that must resolve compatibility issues at runtime, which also results in the weakly typed programming of analytics. Additionally, the UIMA's native indexing of annotation types will no longer work with such a type system, which prompts an additional programming effort from developers.

The aforementioned examples suggest that establishing a single type system that could be shared among all providers is unlikely to ever take place due to the variability in requirements and applicability. Instead, we adopt an idea of using a *conversion* mechanism that enables aligning types across type systems. The conversion has commonly been solved programmatically by creating UIMA analytics that map all or (more likely) selected types between two type systems. For instance, U-Compare features a component that translates some of the CPP types into the U-Compare types. The major drawback of such a solution is the necessity of having to implement an analytic which requires programming skills and becomes an arduous task with an increasing number of type systems. In contrast, we propose a conversion based *entirely* on developers' writing a query in the well established SPARQL language,




(a) cTAKES (b) ACE

(c) Events Type System

Figure 2: UML diagrams representing fragments of type systems that show differences in encoding event structures.

an official W3C Recommendation[2]. Our approach involves 1) the serialisation of UIMA's internal data structures to RDF[3], 2) the execution of a user-defined, type-conversion SPARQL query, and 3) the deserialisation of the results back to the UIMA structure.

The remainder of this paper is organised as follows. The next section presents related work. Section 3 provides background information on UIMA, RDF and SPARQL. Section 4 discusses the proposed representation of UIMA structures in RDF, whereas Section 5 examines the utility of our method. Section 6 details the available implementation, and Section 7 concludes the paper.

## 2 Related Work

In practice, type alignment or conversion is the creation of new UIMA feature structures based on the existing ones. Current efforts in this area mostly involve solutions that are essentially (cascaded) finite state transducers, i.e., an input stream of existing feature structures is being matched against developers' defined patterns, and if a match is found, a series of actions follows and results in one or more output structures.

TextMarker (Kluegl et al., 2009) is currently one of the most comprehensive tools that defines its own rule-based language. The language capabilities include the definition of new types, annotation-based regular expression matching and a rich set of condition functions and actions. Combined with a built-in lexer that produces basic token annotations, TextMarker is essentially a self-contained, UIMA-based annotation tool.

Hernandez (2012) proposed and developed a suite of tools for tackling the interoperability of components in UIMA. The suite includes uima-mapper, a conversion tool designed to work with a rule-based language for mapping UIMA annotations. The rules are encoded in XML, and–contrary to the previous language that relies solely on its own syntax—include XPath expressions for patterns, constraints, and assigning values to new

[2]http://www.w3.org/TR/2013/REC-sparql11-overview-20130321

[3]http://www.w3.org/RDF/

feature structures. This implies that the input of the conversion process must be encoded in XML.

PEARL (Pazienza et al., 2012) is a language for projecting UIMA annotations onto RDF repositories. Similarly to the previous approaches, the language defines a set of rules triggered upon encountering UIMA annotations. The language is designed primarily to work in CODA, a platform that facilitates population of ontologies with the output of NLP analytics. Although it does not directly facilitate the production or conversion of UIMA types, the PEARL language shares similarities to our approach in that it incorporates certain RDF Turtle, SPARQL-like semantics.

Contrary to the aforementioned solutions, we do not define any new language or syntax. Instead, we rely completely on an existing data query and manipulation language, SPARQL. By doing so, we shift the problem of conversion from the definition of a new language to representing UIMA structures in an existing language, such that they can be conveniently manipulated in that language.

A separate line of research pertains to the formalisation of textual annotations with knowledge representations such as RDF and OWL[4]. Buyko *et al.* (2008) link UIMA annotations to the reference ontology OLiA (Chiarcos, 2012) that contains a broad vocabulary of linguistic terminology. The authors claim that two conceptually similar type systems can be aligned with the reference ontology. The linking involves the use of OLiA's associated annotation and linking ontology model pairs that have been created for a number of annotation schemata. Furthermore, a UIMA type system has to define additional features for each linked type that tie a given type to an annotation model. In effect, in order to convert a type from an arbitrary type system to another similar type system, both systems must be modified and an annotation and linking models must be created. Such an approach generalises poorly and is unsuitable for impromptu type system conversions.

## 3 Background

### 3.1 UIMA Overview

UIMA defines both structures and interfaces to facilitate interoperability of individual processing components that share type systems. Type systems may be defined in or imported by a processing component that produces or modifies annotations

[4]http://www.w3.org/TR/owl2-overview/

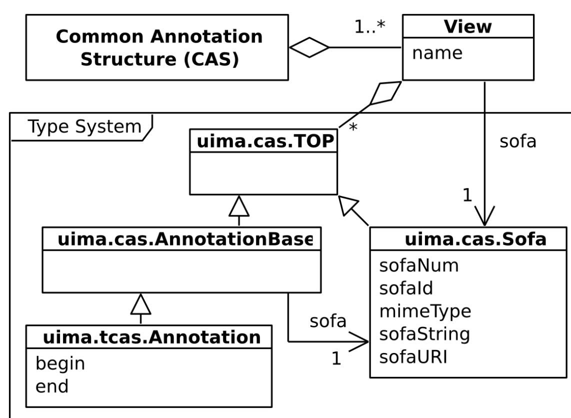



Figure 3: UML diagram representing relationships between CASes, views, and feature structures in UIMA. The shown type system is a fragment of the built-in UIMA type system.

in a *common annotation structure* (CAS), i.e., a CAS is the container of actual data bound by the type system.

Types may define multiple primitive *features* as well as references to *feature structures* (data instances) of other types. The single-parent inheritance of types is also possible. The resulting structures resemble those present in modern object-oriented programming languages.

Feature structures stored in a CAS may be grouped into several *views*, each of which having its own *subject of analysis* (Sofa). For instance, one view may store annotations about a Sofa that stores an English text, whereas another view may store annotations about a different Sofa that stores a French version of the same text. UIMA defines built-in types including primitive types (boolean, integer, string, etc.), arrays, lists, as well as several complex types, e.g., `uima.tcas.Annotation` that holds a reference to a Sofa the annotation is asserted about, and two features, begin and end, for marking boundaries of a span of text. The relationships between CASes, views, and several prominent built-in types are shown in Figure 3.

The built-in complex types may further be extended by developers. Custom types that mark a fragment of text usually extend `uima.tcas.Annotation`, and thus inherit the reference to the subject of analysis, and the begin and end features.

| UIMA element/representation | RDF resource |
|---|---|
| CAS | <uima:aux:CAS> |
| Access to CAS's views | rdfs:member or rdf:_1, rdf:_2, ... |
| View | <uima:aux:View> |
| View's name | <uima:aux:View:name> |
| View's Sofa | <uima:aux:View:sofa> |
| Access to view's feature structures | rdfs:member or rdf:_1, rdf:_2, ... |
| Access to feature structure's sequential number | <uima:aux:seq> |
| Type uima.tcas.Annotation | <uima:ts:uima.tcas.Annotation> |
| Feature uima.tcas.Annotation:begin | <uima:ts:uima.cas.Annotation:begin> |
| Access to uima.cas.ArrayBase elements | rdfs:member or rdf:_1, rdf:_2, ... |

Table 1: UIMA elements and their corresponding RDF resource representations

### 3.2 RDF and SPARQL

Resource Description Framework (RDF) is a method for modeling concepts in form of making statements about resources using triple subject-predicate-object expressions. The triples are composed of *resources* and/or *literals* with the latter available only as objects. Resources are represented with valid URIs, whereas literals are values optionally followed by a datatype. Multiple interlinked subject and objects ultimately constitute *RDF graphs*.

SPARQL is a query language for fetching data from RDF graphs. Search patterns are created using RDF triples that are written in RDF Turtle format, a human-readable and easy to manipulate syntax. A SPARQL triple may contain *variables* on any of the three positions, which may (and usually does) result in returning multiple triples from a graph for the same pattern. If the same variable is used more than once in patterns, its values are bound, which is one of the mechanisms of constraining results.

Triple-like patterns with variables are simple, yet expressive ways of retrieving data from an RDF graph and constitute the most prominent feature of SPARQL. In this work, we additionally utilise features of SPARQL 1.1 Update sublanguage that facilitates graph *manipulation*.

## 4 Representing UIMA in RDF

We use RDF Schema[5] as the primary RDF vocabulary to encode type systems and feature structures in CASes. The schema defines resources such as rdfs:Class, rdf:type (to denote a membership of an instance to a particular class) and rdfs:subClassOf (as a class inheritance property)[6]. It is a popular description language for expressing a hierarchy of concepts, their instances and relationships, and forms a base for such semantic languages as OWL.

The UIMA type system structure falls naturally into this schema. Each type is expressed as rdfs:Class and each feature as rdfs:Property accompanied by appropriate rdfs:domain and rdfs:range statements. Feature structures (instances) are then assigned memberships of their respective types (classes) through rdf:type properties.

A special consideration is given to the type ArrayBase (and its extensions). Since the order of elements in an array may be of importance, feature structures of the type ArrayBase are also instances of the class rdf:Seq, a sequence container, and the elements of an array are accessed through the properties rdf:_1, rdf:_2, etc., which, in turn, are the subproperties of rdfs:member. This enables querying array structures with preserving the order of its members. Similar, enumeration-property approach is used for views that are members of CASes and feature structures that are members of views. The order for the latter two is defined in the internal indices of a CAS and follows the order in which the views and feature structures were added to those indices.

We also define several auxiliary RDF resources to represent relationships between CASes, views and feature structures (cf. Figure 3). We introduced the scheme name "uima" for the URIs of

[5] http://www.w3.org/TR/rdf-schema/

[6] Following RDF Turtle notation we denote prefixed forms of RDF resources as prefix:suffix and their full forms as <fullform>

```
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
INSERT {
  _:newSentence a <uima:ts:our.Sentence> ;
    <uima:ts:uima.tcas.Annotation:begin> ?begin ;
    <uima:ts:uima.tcas.Annotation:end> ?end .
  ?view rdfs:member _:newSentence .
}
WHERE {
  ?view a <uima:aux:View> ;
    rdfs:member ?sentence .
  ?sentence a <uima:ts:their.Sentence> ;
    <uima:ts:uima.tcas.Annotation:begin> ?begin ;
    <uima:ts:uima.tcas.Annotation:end> ?end .
}
```

Figure 4: Complete SPARQL query that converts the sentence type in one type system to a structurally identical type in another type system.

the UIMA-related resources. The fully qualified names of UIMA types and their features are part of the URI paths. The paths are additionally prefixed by "`ts:`" to avoid a name clash against the aforementioned auxiliary CAS and view URIs that, in turn, are prefixed with "`aux:`". Table 1 summarises most of the UIMA elements and their corresponding representations in RDF.

## 5 Conversion Capabilities

In this section we examine the utility of the proposed approach and the expressiveness of SPARQL by demonstrating several conversion examples. We focus on technical aspects of conversions and neglect issues related to a loss or deficiency of information that is a result of differences in type system conceptualisation (as discussed in Introduction).

### 5.1 One-to-one Conversion

We begin with a trivial case where two types from two different type systems have exactly the same names and features; the only difference lies in the namespace of the two types. Figure 4 shows a complete SPARQL query that converts (copies) `their.Sentence` feature structures to `our.Sentence` structures. Both types extend the `uima.tcas.Annotation` type and inherit its begin and end features. The WHERE clause of the query consists of patterns that match CASes' views and their feature structures of the type `their.Sentence` together with the type's begin and end features.

For each solution of the WHERE clause (each retrieved tuple), the INSERT clause then creates a new sentence of the target type `our.Sentence` (the `a` property is the shortcut of `rdf:type`)

```
INSERT {
?eventUri a gen:NamedEvent ;
  gen:NamedEvent:name ?type ;
  tcas:Annotation:begin ?anchorBegin ;
  tcas:Annotation:end ?anchorEnd ;
  gen:Event:participants ?arrayUri .
?arrayUri a cas:FSArray, rdf:Seq ;
  ?argumentIdxUri _:participant .
_:participant a gen:NamedEventParticipant ;
  gen:NamedEventParticipant:name ?role ;
  gen:EventParticipant:target ?target .
?view rdfs:member ?eventUri .
}
WHERE {
?view a aux:View ;
  rdfs:member ?event .
?event a ace:Event ;
  ace:Event:type ?type ;
  ace:Event:mentions ?mentions .
?mentions rdfs:member ?mention .
?mention a ace:EventMention ;
  ace:EventMention:arguments ?arguments ;
  ace:EventMention:anchor ?anchor ;
  aux:seq ?mentionSeq .
?anchor tcas:Annotation:begin ?anchorBegin ;
  tcas:Annotation:end ?anchorEnd .
?arguments rdfs:member ?argument ;
  ?argumentIdxUri ?argument .
?argument ace:EventMentionArgument:role ?role ;
  ace:EventMentionArgument:target ?target ;
  aux:seq ?argumentSeq .

BIND(URI(CONCAT("tmp:", STR(?mentionSeq)))
    AS ?eventUri)
BIND(URI(CONCAT("tmp:", STR(?mentionSeq), "#array"))
    AS ?arrayUri)
}
```

Figure 5: SPARQL query that aligns different conceptualisations of event structures between two type systems. Prefix definitions are not shown.

and rewrites the begin and end values to its features. The *blank node* `_:sentence` is going to be automatically re-instantiated with a unique resource for each matching tuple making each sentence node distinct. The last line of the INSERT clause ties the newly created sentence to the view, which is UIMA's equivalent of indexing a feature structure in a CAS.

### 5.2 One-to-many Conversion

In this use case we examine the conversion of a container of multiple elements to a set of disconnected elements. Let us consider event types from the ACE and Events type systems as shown in Figures 2(b) and 2(c), respectively. A single `Event` structure in the ACE type system aggregates multiple `EventMention` structures in an effort to combine multiple text evidence supporting the same event. The `NamedEvent` type in the Events type system, on the other hand, makes no such provision and is agnostic to the fact that multiple mentions may refer to the same event.

To avoid confusion, we will refer to the types using their RDF prefixed notations, "`ace:`" and "`gen:`", to denote the ACE and "generic" Events type systems, respectively.

The task is to convert all `ace:Events` and their `ace:EventMentions` into `gen:NamedEvents`. There is a couple of nuances that need to be taken into consideration. Firstly, although both `ace:EventMention` and `gen:NamedEvent` extend `uima.tcas.Annotation`, the begin and end features have different meanings for the two event representations. The `gen:NamedEvent`'s begin and end features represent an anchor/trigger, a word in the text that initiates the event. The same type of information is accessible from `ace:EventMention` via its `anchor` feature instead. Secondly, although it may be tempting to disregard the `ace:Event` structures altogether, they contain the `type` feature whose value will be copied to `gen:NamedEvent`'s `name` feature.

The SPARQL query that performs that conversion is shown in Figure 5. In the WHERE clause, for each `ace:Event`, patterns select `ace:EventMentions` and for each `ace:EventMention`, `ace:EventMentionArguments` are also selected. This behaviour resembles triply nested *for* loop in programming languages. Additionally, `ace:Event`'s type, `ace:EventMention`'s anchor begin and end values, and `ace:EventMentionArgument`'s role and target are selected. In contrast to the previous example, we cannot use blank nodes for creating event resources in the INSERT clause, since the retrieved tuples share event URIs for each `ace:EventMentionArgument`. Hence the last two `BIND` functions create URIs for each `ace:EventMention` and its array of arguments, both of which are used in the INSERT clause.

Note that in the INSERT clause, if several `gen:NamedEventParticipants` share the same `gen:NamedEvent`, the definition of the latter will be repeated for each such participant. We take advantage of the fact that adding a triple to an RDF graph that already exists in the graph has no effect, i.e., an insertion is simply ignored and no error is raised. Alternatively, the query could be rewritten as two queries, one that creates

```
INSERT {
?mentionUri a ace:EntityMention ;
  tcas:Annotation:begin ?begin ;
  tcas:Annotation:end ?end ;
  ace:EntityMention:type ?type ;
  ace:EntityMention:role ?relation .
?mentions a cas:FSArray, rdf:Seq  ;
  ?mentionMemberUri ?mentionUri .
?entityUri a ace:Entity ;
  ace:Entity:mentions ?mentions .
?view rdfs:member ?entityUri .
}
WHERE {
?view a <uima:View> ;
  rdfs:member ?chain .
?chain a dkpro:CoreferenceChain ;
  dkpro:CoreferenceChain:first/
    dkpro:CoreferenceLink:next* ?link ;
  <uima:aux:seq> ?chainSeq .
?link <uima:aux:seq> ?linkSeq ;
  dkpro:CoreferenceLink:referenceType ?type ;
  dkpro:CoreferenceLink:referenceRelation ?relation ;
  tcas:Annotation:begin ?begin ;
  tcas:Annotation:end ?end .

BIND(URI(CONCAT("entity:", STR(?chainSeq)))
    AS ?entityUri)
BIND(URI(CONCAT("mention:", STR(?linkSeq)))
    AS ?mentionUri)
BIND(URI(CONCAT(STR(?entityUri), "#mentions"))
    AS ?mentions)
BIND(URI(CONCAT(str(rdf:), "_", str(?linkSeq)))
    AS ?mentionMemberUri) .
}
```

Figure 6: SPARQL query that converts coreferences expressed as linked lists to an array representation. Prefix definitions are not shown.

`gen:NamedEvent` definitions and another that creates `gen:NamedEventParticipant` definitions.

To recapitulate, RDF and SPARQL support one-to-many (and many-to-one) conversions by storing only unique triple statements and by providing functions that enable creating arbitrary resource identifiers (URIs) that can be shared between retrieved tuples.

### 5.3 Linked-list-to-Array Conversion

For this example, let us consider two types of structures for storing coreferences from the DKPro and ACE type systems, as depicted in Figures 1(a) and 1(c), respectively.

The idea is to convert DKPro's chains of links into ACE's entities that aggregate entity mentions, or—using software developers' vocabulary—to convert a linked list into an array. The SPARQL query for this conversion is shown in Figure 6.

The WHERE clause first selects all `dkpro:CoreferenceChain` instances from views. Access to `dkpro:CoreferenceLink` instances for each chain is provided by a *property*

*path*. Property paths are convenient shortcuts for navigating through nodes of an RDF graph. In this case, the property path expands to the chain's `first` feature/property followed by any number (signified by the asterisk) of links' `next` feature/property. The pattern with this path will result in returning all links that are accessible from the originating chain; however, according to the SPARQL specification, the order of links is not guaranteed to be preserved, which in coreference-supporting applications is usually of interest. A solution is to make use of the property `<uima:aux:seq>` that points to the sequential number of a feature structure and is unique in the scope of a single CAS. Since feature structures are serialised into RDF using deep-first traversal, the consecutive link structures for each chain will have their sequence numbers monotonically increasing. These sequence numbers are translated to form `rdf:_nn` properties (*nn* standing for the number), which facilitates the order of elements in the `ace:Entity` array of mentions[7]. It should be noted, however, that using the sequence number property will work only if the links of a chain are not referred to from another structure. There is another, robust solution (not shown due to space limitation and complexity) that involves multiple INSERT queries and temporary, supporting RDF nodes. RDF nodes that are not directly relevant to a CAS and its feature structures are ignored during the deserialisation process, and thus it is safe to create any number of such nodes.

## 6 Tool Support

We have developed a UIMA analysis engine, SPARQL Annotation Editor, that incorporates the serialisation of a CAS into RDF (following the protocol presented in Section 4), the execution of a user-defined SPARQL query, and the deserialisation of the updated RDF graph back to the CAS. The RDF graph (de)serialisation and SPARQL query execution is implemented using Apache Jena[8], an open-source framework for building Semantic Web applications.

To further assist in the development of type-conversion SPARQL queries, we have provided two additional UIMA components, RDF Writer and RDF Reader. RDF Writer serialises CASes to files that can then be used with SPARQL query engines, such as Jena Fuseki (part of the Apache Jena project), to develop and test conversion queries. The modified RDF graphs can be imported back to a UIMA application using RDF Reader, an RDF deserialisation component.

The three components are featured in Argo (Rak et al., 2012), a web-based workbench for building and executing UIMA workflows.

## 7 Conclusions

The alignment of types between different type systems using SPARQL is an attractive alternative to existing solutions. Compared to other solutions, our approach does not introduce a new language or syntax; to the contrary, it relies entirely on a well-defined, standardised language, a characteristic that immediately broadens the target audience. Likewise, developers who are unfamiliar with SPARQL should be more likely to learn this well-maintained and widely used language than any other specialised and not standardised syntax.

The expressiveness of SPARQL makes the method superior to the rule-based techniques, mainly due to SPARQL's inherent capability of random data access and simple, triple-based querying. At the same time, the semantic cohesion of data is maintained by a graph representation.

The proposed solution facilitates the rapid alignment of type systems and increases the flexibility in which developers choose processing components to build their UIMA applications. As well as benefiting the design of applications, the conversion mechanism may also prove helpful in the development of components themselves. To ensure interoperability, developers usually adopt an existing type system for a new component. This essential UIMA-development practice undeniably increases the applicability of such a component; however, at times it may also result in having the ill-defined representation of the data produced by the component. The availability of an easy-to-apply conversion tool promotes constructing fine-tuned type systems that best represent such data.


## Acknowledgments

This work was partially funded by the MRC Text Mining and Screening grant (MR/J005037/1).


[7] The `rdf:_nn` properties are not required to be consecutive in an RDF container

[8] http://jena.apache.org/

## References

W A Baumgartner, K B Cohen, and L Hunter. 2008. An open-source framework for large-scale, flexible evaluation of biomedical text mining systems. *Journal of biomedical discovery and collaboration*, 3:1+.

E Buyko, C Chiarcos, and A Pareja-Lora. 2008. Ontology-based interface specifications for a nlp pipeline architecture. In *Proceedings of the Sixth International Conference on Language Resources and Evaluation (LREC'08)*, Marrakech, Morocco.

C Chiarcos. 2012. Ontologies of linguistic annotation: Survey and perspectives. In *Proceedings of the Eighth International Conference on Language Resources and Evaluation (LREC'12)*, pages 303–310.

D Ferrucci and A Lally. 2004. UIMA: An Architectural Approach to Unstructured Information Processing in the Corporate Research Environment. *Natural Language Engineering*, 10(3-4):327–348.

Iryna Gurevych, Max Mühlhäuser, Christof Müller, Jürgen Steimle, Markus Weimer, and Torsten Zesch. 2007. Darmstadt Knowledge Processing Repository Based on UIMA. In *Proceedings of the First Workshop on Unstructured Information Management Architecture at Biannual Conference of the Society for Computational Linguistics and Language Technology*, Tübingen, Germany.

U Hahn, E Buyko, R Landefeld, M Mühlhausen, M Poprat, K Tomanek, and J Wermter. 2008. An Overview of JCORE, the JULIE Lab UIMA Component Repository. In *Proceedings of the Language Resources and Evaluation Workshop, Towards Enhanc. Interoperability Large HLT Syst.: UIMA NLP*, pages 1–8.

N Hernandez. 2012. Tackling interoperability issues within UIMA workflows. In *Proceedings of the Eight International Conference on Language Resources and Evaluation (LREC'12)*, Istanbul, Turkey. European Language Resources Association (ELRA).

P Kluegl, M Atzmueller, and F Puppe. 2009. TextMarker: A Tool for Rule-Based Information Extraction. In *Proceedings of the Biennial GSCL Conference 2009, 2nd UIMA@GSCL Workshop*, pages 233–240. Gunter Narr Verlag.

M T Pazienza, A Stellato, and A Turbati. 2012. PEARL: ProjEction of Annotations Rule Language, a Language for Projecting (UIMA) Annotations over RDF Knowledge Bases. In *Proceedings of the Eight International Conference on Language Resources and Evaluation (LREC'12)*, Istanbul, Turkey. European Language Resources Association (ELRA).

R Rak, A Rowley, W Black, and S Ananiadou. 2012. Argo: an integrative, interactive, text mining-based workbench supporting curation. *Database : The Journal of Biological Databases and Curation*, page bas010.

G K Savova, J J Masanz, P V Ogren, J Zheng, S Sohn, K C Kipper-Schuler, and C G Chute. 2010. Mayo clinical Text Analysis and Knowledge Extraction System (cTAKES): architecture, component evaluation and applications. *Journal of the American Medical Informatics Association : JAMIA*, 17(5):507–513.

P Thompson, Y Kano, J McNaught, S Pettifer, T K Attwood, J Keane, and S Ananiadou. 2011. Promoting Interoperability of Resources in META-SHARE. In *Proceedings of the IJCNLP Workshop on Language Resources, Technology and Services in the Sharing Paradigm (LRTS)*, pages 50–58.

K Verspoor, W Baumgartner Jr, C Roeder, and L Hunter. 2009. Abstracting the Types away from a UIMA Type System. *From Form to Meaning: Processing Texts Automatically.*, pages 249–256.