Annotation Process Management Revisited

Dain Kaplan, Ryu Iida, Takenobu Tokunaga

Department of Computer Science, Tokyo Institute of Technology
{dain,ryu-i,take}@cl.cs.titech.ac.jp

Abstract

Proper annotation process management is crucial to the construction of corpora, which are indispensable to the data-driven techniques that have come to the forefront in NLP during the last two decades. This paper first raises a list of 10 needs that any general purpose annotation system should address, such as user & role management, delegation & monitoring of work, diffing annotators' work, versioning of corpora, multilingual support, and so on. A framework to address these needs is then proposed. The explanation of the framework is followed by an introduction of SLATE (Segment and Link-based Annotation Tool *Enhanced*), the second iteration of a web-based annotation tool, which is being rewritten to implement the proposed framework.

1. Introduction

Corpus-based approaches have risen in popularity over the last two decades in the field of natural language processing (NLP); in many areas corpus-based approaches equal in performance, or rival traditional rule-based methods. With the increase in computational power and the advancement and ease of use of machine-learning (ML) techniques, it is no wonder that corpus-based approaches continue to gain in popularity. In the late 1990s a sizable textbook (Manning and Schuetze, 1999) was published attesting to their merits, and even a decade before, research had appeared pursuing such techniques (Charniak, 1993). Corpus-based approaches also allow for more language and domain independence within a model, important in today's multilingual world.

The advent of corpus-based approaches also meant that the creation of corpora was required — corpus-based approaches are obviously impossible without them. The individual creation of custom tools for annotation tasks is a tremendous investment of time and labor, which when viewed in a wholistic manner shows how repetitive and wasteful such an investment can be. These custom fitted tools do of course address the demands of the project for which they were born, but also because of this tend towards inflexibility and disposability. Regardless of the problems of interoperability that arise when different tools are used with different data formats (something desirable when trying to cross-test methods with different pre-existing datasets), a cycle forms of creating new tools for trivial tasks, or perhaps even worse conforming and compromising annotation criteria to fit within the limits of an existing tool made for a previous project, which impinges on the quality of the resource. Corpus-based methods will only grow in scale and complexity, and so it is crucial the annotation tool does not stand in the way.[1]

The bottom line is that corpus creation (and also management) is time consuming enough without having to spend more resources in battling the development of a custom tool. In addition, previous studies have shown that the early phases of a project are the most volatile (Marcus et al.,

1993), resulting in rapid changes to the annotation schema. This can also cause delays and cost additional resources if the annotation tool is too rigid or otherwise unable to adapt to the changes.

A survey (Dipper et al., 2004) proposed seven categories desirable for any annotation tool: diversity of data, multi-level annotation, diversity of annotation, simplicity, customizability, quality assurance and convertibility. The goal of these categories is to remove the obstacles outlined as problems above. They are well thought out, but there is one caveat to them: they are document-centric, or rather, they simply do not address the bigger scope of managing the annotation process. Furthermore, though they do raise concerns about areas that need to be addressed, they do not propose how these areas are to be tackled. In simple corpora this may not be much of a concern, but in larger projects – and as corpus-based techniques continue to grow and advance so do the sizes of the corpora (Davies, 2009) – it becomes a serious issue.

In order to create a tool suitable for use in the future, it is important to pinpoint the needs of an annotation project. In Section 3. we outline what we think these needs are, and then in Section 4. propose a framework for addressing them. We next introduce SLATE (Segment and Link-based Annotation Tool *Enhanced*), the second iteration of a web-based annotation tool under development that utilizes this framework, geared towards being a possible solution. We then conclude the paper and discuss future work.

2. Related Work

Over the years countless custom annotation tools have been created to meet specific demands of various annotation projects; more recent examples include (Stührenberg et al., 2007; Russell et al., 2005). However, many more never see the public light of day, because they were created behind the scenes hastily and then disposed of. They were mostly created because a tool did not exist that was capable of providing the feature set needed for development of the corpus within a reasonable amount of time.[2] A recent example includes Serengeti (Stührenberg et al., 2007), which has been

---

[1] In addition, as data sets continue to grow in size and methods mature, it will be necessary to compare them, meaning a standardized format will also be necessary.

[2] Time to installation, learning curve to use the software, time to understand if the tool can meet the project's needs, etc. are all factors in selecting an annotation tool.

developed for the specific task of anaphoric relations and lexical-chains. Such solutions, however, are not generic; in other words, though they may be quite ideal for one annotation task, they may not readily adapt to other tasks, not to mention the difficulties in data interoperability/interchange. The problems with task-oriented annotation tools will grow as corpus-based techniques do; a unified framework must exist to allow the corpora to work with one another.

Some general-purpose annotation tools have also appeared over the years (Asan and Orăsan, 2003; Cunningham et al., 2002; Dennis et al., 2003; Mueller and Strube, 2001; Callisto, 2002). Some of these are more extensible than others, some are discontinued, some include advanced features for processing data for semi-automatic annotation. PALinkA (Asan and Orăsan, 2003) might be the easiest of these to use, as it requires little setup, but has not seen an update in more than four years and does not support many of the key features we propose are mandatory for annotation software moving forward. As corpus-based techniques continue to flourish, allowing for layered annotations on the same base dataset will be more crucial.

There is an existing framework for dealing with annotations called ATLAS (Flexible and Extensible Architecture for Linguistic Annotation) (Bird et al., 2000); but it is highly generalized, and becomes quite complex for the rather simple task of textual annotation (that always involves spanning from one offset in a document to another). Further, ATLAS is concerned only with annotations, not with the broader picture of annotation process management. Thus we have opted to base our approach on Segments and Links (Takahashi and Inui, 2006; Noguchi et al., 2008), geared towards text annotation; it is introduced with our extensions in Section 4..

3. Annotation Needs

Creating corpora is becoming a serious endeavor; it is an ordeal entirely separate from, and from some viewpoints secondary to, the technique that will utilize a given language resource. Such a trend will only worsen as techniques, and therefore the corpora they use, continue to grow in scale and complexity.

Thus it is not only that the annotation system must be flexible enough to accommodate a wide variety of annotation tasks, but that it must also provide for a means to *manage* the tasks themselves. Otherwise, the creation of the resources will hinder the development of the technique, the reason behind why they were made.

Let us concretize the definitions of terms important for defining annotation needs: annotation tasks, annotation projects, and annotation systems. An annotation task has a specific goal in mind, such as annotating all predicate-argument dependencies in a set of raw resources, or identifying all coreference-chains in a collection of news articles, etc. An annotation task is more generally a *type* of annotation work to be done, whereas an annotation project, is an *instance* of that work. For example, if we wish to perform two tasks on the same dataset, such as predicate-argument dependencies and then coreference-chains, we should consider these as two separate annotation projects, as well. An annotation system is software that allows a user to create

annotation projects for annotation tasks.

Let us then specify what is needed from an annotation system. (Dipper et al., 2004) proposed seven categories for what is needed from an annotation tool, but they operate mostly at the document-level. Our list of needs below focuses instead at a more macro, annotation project management level. We therefore skip needs related only to the act of annotation (such as appearance), though it is covered by our framework introduced in Section 4..

(1) **User and role management**. Annotation projects are too complex to have a single user type with no roles assigned to them. The system must distinguish between annotators and administrators in order to prevent annotators from unwittingly altering a key part of the project they should not have access to in the first place.

(2) **Delegation and monitoring of work**. The system must allow for an administrator to assign/reassign work to annotators, and to monitor their progress. It is important to forestall an annotator falling behind due to difficulties or other factors as it may also delay the completion of the corpus if the work is not reassigned, or the obstacle stopping the annotator not resolved.

(3) **Adaptability to new annotation tasks**. The system must be flexible enough to easily accommodate a new annotation task. If administrators cannot easily create a new project, and define the annotation requirements then the system will not be useful to them.

(4) **Adaptability within the current annotation task**. During the lifespan of any given project, it often meets with many changes, especially during the initial phases of a project. It is crucial that the system allow for the adjustment of annotation guidelines in such a way as to (1) facilitate the correction of any already annotated items by identifying those resources, and (2) to have the flexibility to accommodate the changes themselves.

(5) **Diffing and merging**. Creating a corpus often entails the *diffing and merging* of data from multiple annotators on a single resource to create a gold standard. Annotator agreement is an important statistic in general for corpora, but in order to have an end product, it may be important to resolve any differences between different annotators. The system must allow an administrator to diff all annotated resources with multiple sets of annotations, and then to merge any differences that are found. In the event that multiple sets of annotations on the same resource are desired however, the system should also not stand in the way.

(6) **Versioning of corpora**. A corpus is a product, an end result. But just as any other product goes through life cycles, so too may the corpus. In other words, after all work has been completed (and the gold standard created), there must be a way to package the result and label it uniquely for use. After a release there are often fixes, amendments, etc. and these changes must be

tracked so that an additional version can be released. Without management of versioning, the "current state" of the annotation project is all that is known; for large projects, especially, it is important to identity milestones or to "tag" a given state so that one may be able to go back to it later. This way changes made after the "tagging" will not unknowingly be included into a release.
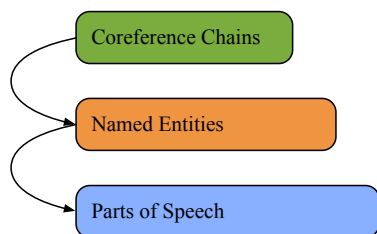

Figure 1: Higher layers access lower layers as a base for more stringent annotation

(7) **Extensibility in terms of layering**. As corpora are continuing to grow in size, it is no wonder that they are also fundamentally becoming more complex. We have seen recently the stacking, or layering, of corpora upon one another. Attempts at diversifying a single corpus with various types of annotations are on the rise, such as the Discourse TreeBank on top of the Penn TreeBank (Miltsakaki et al., 2004), or the NAIST Text Corpus atop the Kyoto Text Corpus (Iida et al., 2007). The system must allow for adding new layers upon previous ones, seamlessly. Often the lower layers are directly referenced by the new ones. It must allow for this as well *without* jeopardizing the quality of the lower levels. This idea is shown in Figure 1, where if we consider each level being a separate project, the one above directly references annotations in the preceding layer.

(8) **Extensibility in terms of tools**. Many annotation tasks these days involve (semi-)automatic tagging, followed by an annotator reviewing and correcting any mistakes in the output. The system should ideally allow the annotator to call plugins to the system to tag a given resource to facilitate this. This step could be done prior to data import, but allowing the user to do so afterwards has several benefits, such as enabling them to quickly view the results of the automatic tagging, and to rerun it with different parameters again should it not have provided the expected output. The plugins should be able to be created by anyone (not only the creators of the system).

(9) **Extensibility in terms of importing/exporting**. The system should also allow the user to either define rules or call plugins to convert the data from one format to another. This allows the system to be somewhat agnostic to the data format. This is important as there are a variety of formats, though the system should provide its own that is capable of handling any supported annotation scheme. Including generated comments may facilitate in human verification of the exported resource.

This step could be done after export, but having the ability to have the data processed automatically increases productivity for menial tasks that should not require human intervention (and which may also introduce error).

(10) **Support for multiple languages**. Research today is carried out in a variety of languages, and the system should support them.

4. Framework Overview

The ten items listed above can be subdivided into those that require a framework for support, and those that rely on an implementation of a given framework. Without a proper framework in place, (1) - (7) are impossible; (8) - (10) reside entirely with the implementation. We address needs (1) & (2) (User & Project Management, and work delegation) in Section 4.1., (3) & (4) (Adaptability to current and future annotation tasks) in Sections 4.3. and 4.4., and need (7) in Section 4.6.. Needs (5) & (6) require the necessary data within a framework, but also rely heavily on the implementation to supply such features; they appear as the topic is relevant, below. Figure 2 shows a simplified representation in UML of some of the more major entities of the proposed framework. It may be beneficial to refer to it while reading through the explanation.

4.1. User & Project Management

A large part of proper project management involves proper user management (needs (1) & (2)). It is important to encapsulate user responsibilities in a way that facilitates productivity, rather than hinders it due to complexity. User roles are nothing new to software systems, but choosing the right granularity can be tough. We think it is best to keep things as simple as possible while maintaining the needed functionality; thus, we propose having minimally two types of users: administrators and annotators. (Note that user roles are not present in Figure 2.)

An administrator oversees projects, configures them, adds/removes annotators, and assigns annotation tasks. The administrator should also be able to check the progress of the annotators to make sure no roadblocks are preventing them from finishing their assigned work, or if they become incapacitated for some reason, allow the administrator to easily reassign outstanding work. As mentioned in Section 3., the administrator should also be able to create versions of the project (need (6)), import, export, and in all other ways administer the project. They should not, however, be able to annotate resources.

The annotator, therefore, is solely responsible for this task of annotation. They can select a resource that has been assigned to them and annotate it; they are sandboxed from other users to prevent biases from interfering with their work (meaning they do not see others' work).

This separation between administrator and annotator roles is simple, but it is intuitive; it also provides the necessary separation to properly manage a project and still get the annotation work done. Later on an administrator can diff the resources and merge them as necessary for creating a gold standard. More details about project configuration are

explained as the remaining entities and annotation methodology are elaborated on below.
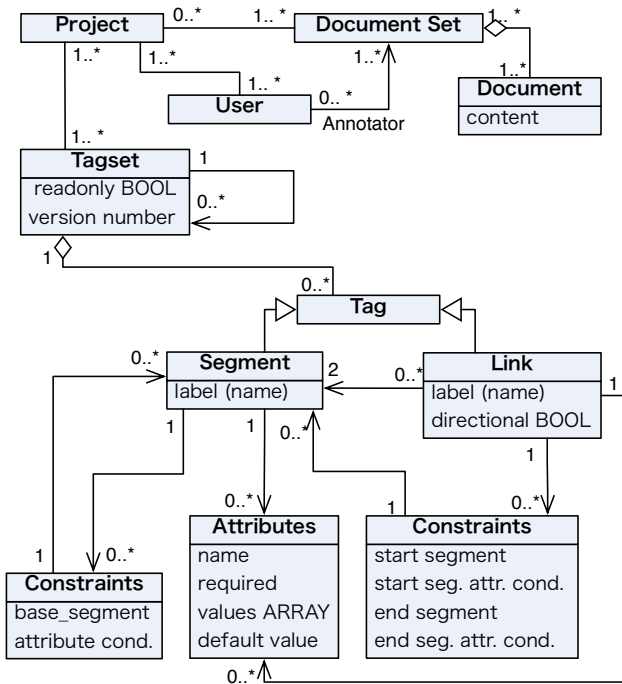

Figure 2: Simplified relationship diagram for entities

4.2. Entities

As Figure 2 shows, there are a number of entities interconnected to one another. In this section we will focus on the more macro-level entities, Project, Document, Document Set, and then touch upon Tagsets, which are explained more in the next section. The framework is based around the concept of a *document*, which is the only entity that can be directly annotated (shown in the upper right corner of Figure 2). It can represent any kind of data you wish (e.g. a news article, paragraph, book). Since dealing directly with large quantities of documents can be daunting, they are grouped into a more macro-level *document set*. A document can be a member of multiple document sets, so a document set could be created to encapsulate both raw resources from different sources, *and* groups of work to delegate to annotators. As the UML figure shows, annotators are assigned document sets, not documents; again, this is to make coordinating large volumes of documents easy, while keeping them coherently managed to some degree.

The highest level entity in the framework is the *project*. A project represents an annotation task, such as predicate-argument dependency or coreference-chains, and contains a reference to one or more document sets. An important point here is that a project contains a *reference to* one or more document sets; the sets themselves exist beyond, or outside of, the project. This means that they can (and should) be reused for multiple projects.

More formally, we define a *user* as an annotator or administrator with access to the system, belonging to one or more projects (a user may be working on multiple tasks, concurrently, or in sequence); an annotator will have access to zero or more document sets within any given project,

while the administrator can do all the actions outlined in Section 4.1.. Since the point of task delegation is to split the work among many individuals, the framework allows for this by enabling the administrator to assign different document sets to different users.

The majority of the UML diagram, however, focuses on *tagsets* and their relations. This is the fundamental mechanism behind generalized annotation in the framework, and is explained in the next section.
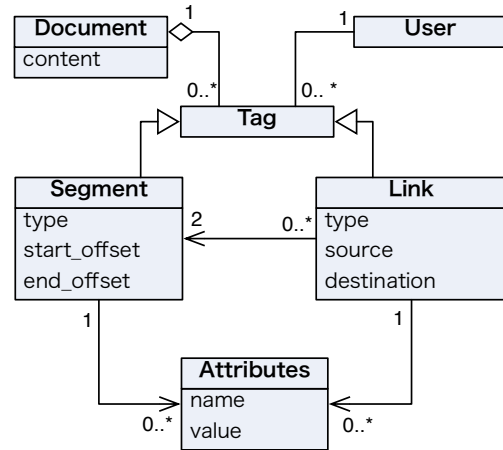

Figure 3: Simplified relationship diagram for annotation instances

4.3. Abstracting Annotation

So far we have outlined several of the main entities that make up the framework. But the crux of being able to support multiple annotation tasks (needs (3) & (4)) is the notion of abstracting annotation. Abstracting annotation means that we remove any definitions for what types of entities can be annotated from the system, and instead create a framework that allows administrators to define them themselves. Conceptually, all an annotation is, is either a label placed on a span of text, or a label placed on a relationship between such spans, as is shown in Figure 4. So we allow the administrator to create as many *definitions* for types of annotations as they like, and then during annotation, the annotator can create *instances* of these admin-defined types, shown in Figures 2 and 3, respectively. It is important to understand there is a difference between a definition of a type, and an instance of that type. Also note that the framework itself is agnostic to the definitions; to the system it sees only different types of segments and links.

The annotations on text-spans we call *segments*, and the relationships between them, *links* (Takahashi and Inui, 2006; Noguchi et al., 2008). Links may be directional, such as in the predicate-argument dependency example shown in Figure 4, or undirected, such as when annotating coreference-chains (see Figure 6). As Figure 2 shows, segments and links both extend from an abstract *tag*. For the remainder of this paper, if segment or link is not specified, then the explanation applies to both.

However, simple labels and their relations are limiting; we might want to store additional information both about segments, and about any links that connect them. For this,
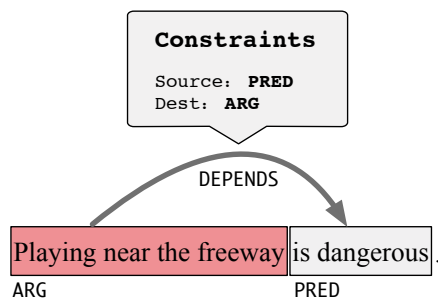
Figure 4: An example showing two segments, and a link between them

attributes are necessary. Say we wish to supplement a segment type definition for "Noun" with information about its plurality; for this we could create an attribute with the possible values of "singular" and "plural". We can then formalize the definition of a tag T to be a label *label* and a set of attributes a_i composed of a name *name* and a value type *val_type*, along with a possible default value *def_val*, and a flag *req* indicating whether or not the attribute is required:

$$T = (label, \{a_0, a_1, \ldots, a_{n-1}\}),$$
$$\text{where } a_i = (name, val\_type, def\_val, req).$$

The value type could be an enumeration, all possible character strings (i.e. free input), number ranges, etc. We can then group these segment and link type definitions into a *tagset*, and assign the tagset to a project. Projects may need to contain multiple tagsets, depending on the task. In the case that the project is extending another previous project (need (7)), it will be important to disable creation of instances from that tagset, and instead only allow the annotator to view them (reflected in Figure 2).

4.4. Tagset Management

Especially during the early phases of a project, the annotation specification is victim to changes in design (need (4)). By creating versions/revisions of tagsets, we can enable the system to keep track of what resources are annotated with which version of each type; allowing an administrator to verify that all documents are annotated with the current spec. Examples include an administrator adding an attribute to verbs, or adding a value to an existing attribute, etc. In the framework all instances of annotations are marked with the version/revision of the tagset that was used when they were annotated. It may also be beneficial to denote the difference between changes that complement a current version (e.g. accommodating a new case found during annotation that does not affect other annotations), i.e. revisions, and changes that require reannotating resources (versions).

4.5. Appearance/Settings

For a visual task like annotation, it is important that as much information as possible is conveyed to the user as quickly as possible. (Dipper et al., 2004) defined the need for customization, and we address this in our framework here (as it is not so much an annotation process level item, it is not included in our list of needs in Section 3.). We define appearance settings for segments/links that can be defined at the tagset, project, and annotator level. This allows the user to decide how the information is conveyed should they have an inclination to override the default. In addition, attributes on specific instances of segments/links may carry importance and affect subsequent annotation (e.g. annotating the verbs that relate to nouns with their plurality attribute set to "plural"). For this reason appearance settings can also be set for specific type definitions that have certain attribute values.

The appearance of the resource and the annotations should be left up to the user. Annotation is a tiring, repetitive process; creating as stress-free an environment as possible for the annotator is therefore key. Preferences should enable the annotator to change the appearance of the tagsets and of the editor in general. A white background can be tiring on the eyes, for example, so the annotator should be able to pick a color that suits him/her. Since segments and links possessing certain attributes can change in color, the user should be allowed to pick the colors that are the most meaning/pleasing to them. Yet, an annotation scheme without any colors is next to meaningless; therefore, we define the order of overriding for appearance settings, shown in Figure 5.
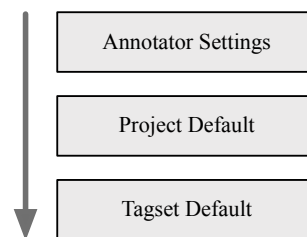

Figure 5: Order of appearance overrides for segments and links; above overrides below

4.6. Annotation

With definitions for the various entities out of the way, we move on to discussing the process of annotation. There is one more feature of the framework that greatly reduces careless annotator errors and at the same time improves productivity: attribute constraints. In a sense, attribute constraints work similarly to appearance overrides explained in Section 4.5.; they are defined along with tag type definitions, but they function at the instance level. Instead of affecting appearance, they affect the creation of instances of certain types. Segment and link constraints vary, so we will begin with segments.

A segment instance of a given type can only be created if it has no constraints, or if all of its constraints are satisfied; constraints for segments operate on two levels. The first level relates to how segments overlap, and the second to the attribute values of the overlapping segments. There are three practical cases for segments overlapping, (1) when the new segment will be contained within an existing segment, (2) the new segment will contain an existing segment, or (3) when the new segment and an existing segment perfectly overlap (they annotate the same text span). For each of these three cases, the constraints can further specify what attribute values the existing segment must possess to allow

the new segment to be created. For example, we may wish to create a named entity (NE) on top of a corpus annotated with noun phrases (NP); we could specify that the NE can only be created if it perfectly overlaps with an existing NP. We could also say that a verb phrase must contain a verb segment, or it cannot be created, or that determiners can only be created inside of an NP, etc.

Similarly, link constraints refer to the link's source and destination segments. This allows a link to be attached only to segments with specific attribute values, such as linking an instance of a segment definition for "verb" with the "transitive" attribute set to "true", to the object it modifies. If the verb instance did not contain the attribute value for transitive, then the constraint would not allow a link to be attached to the verb. Figure 4 shows an example of when a link type requires specific segment types for source/destination, but does not care about any attributes. In this way corpus consistency is enforced by preventing many careless annotator mistakes.

Constraints referring to definition types/attributes can also be nested, meaning a different tagset can be used as a basis for another. This allows us to enforce consistency between different layers of a multi-layered corpus, explained a little more in Section 5. below, and to an extent, to allow specifying workflow.

5. An Example

Let us end the explanation of the framework with an example. Take a large project like the Penn Treebank (Marcus et al., 1993), which is annotated with parts-of-speech and syntactic structure. As a layer on top of this, PropBank (Kingsbury and Palmer, 2002) annotates semantic relations. We can create a project in the framework for each of these. Since they annotate the same resources, we can create documents and document sets to represent the raw resources that both projects can refer to. Next, we would create tagsets for POS-tagging and syntactic structure (with segments for the various parts of speech, and links to encode structure), and include these tagsets in the project for the Treebank. We would then create another tagset for PropBank for annotating predicate-argument dependencies (with segments for the frame's various constituents, e.g. arg0, arg1, pred, etc., and links for showing their relations).

Since the PropBank project sits on top of the Treebank project's data, naturally it will refer to the annotations within the Treebank. We therefore make the PropBank have read-only access to the tagsets for POS and syntactic structure. This allows an annotator to see the previously annotated information (contained in a differed and merged gold standard created by an administrator for the Treebank project), but not to edit or unknowingly corrupt it. Further, by adding constraints to the tagset for PropBank, we can insure that the annotator can only select annotations already present in the Treebank (e.g. the annotator cannot select a segment that is not annotated within the syntactic structure annotations).

6. SLATE

SLATE (Segment and Link-based Annotation Tool *Enhanced*) implements the framework described above. It is the next release of SLAT (Segment and Link-based Annotation Tool) (Noguchi et al., 2008), which is being rewritten from scratch as SLATE to improve performance and user experience. It uses a standoff format , which allows for layered annotations by storing them not as additional markup within a resource, but as meta data stored separately. Aside from the needs provided inherently by the framework ((1)–(4), (7)) We are aiming at supporting needs (5) & (6) (diffing and versioning), and looking into (8) & (9) (extensibility) as well. The system currently supports any left-to-right language (need (10)), and has a multilingual interface as well. It is not fully implemented yet, but slated for an alpha release soon.

A brief explanation of some of the points shown in the screenshots follows. Figure 6 shows the document selection screen an annotator is presented with upon logging in. If the annotator is part of multiple projects, all of them will appear here. Information about the selected project, document set, and document appear to the right. After selecting a document for annotation, the annotator is taken to the screen shown in Figure 7. The left panel is the main annotation panel; during cases in which annotating links to an external resource (another document) are necessary, a split panel (two side-by-side editors) will be available. The right side provides a set of tool panels that show document information, the available tagsets for annotation, and a list of segments/links that exist within the current document.

7. Conclusion

This paper presented a list of 10 needs for annotation systems in order to support corpus-based NLP research moving into the future. The needs fall into two groups: framework-level (needs (1) - (7)), and implementation-level ((8) - (10)). A framework was introduced to tackle the framework-level needs, and a system SLATE built on top of the framework, to handle the implementation-level needs. At its core, the framework provides a mechanism for abstracting annotation and the creation of layered resources through its use of tagsets, containing segments and links with attributes as well as constraints. The framework further provides a foundation for managing annotation resources and multiple annotation tasks. The framework allows an administrator to define a task and then monitor its progress, letting the annotator do what they were intended to do: annotate. Appearance settings allow the annotator to quickly know what types of segments/links they are working with, and the constraints reduce negligent annotation errors (such as not allowing predicate-argument dependency to join two arguments), though the appearance alone helps with this. Future work can be divided into two categories: framework-level, and implementation-level. For the former, it may be beneficial to more concretely specify how corpus versioning should be carried out. For the latter, a number of features are still left unimplemented. We plan to release an alpha version soon.
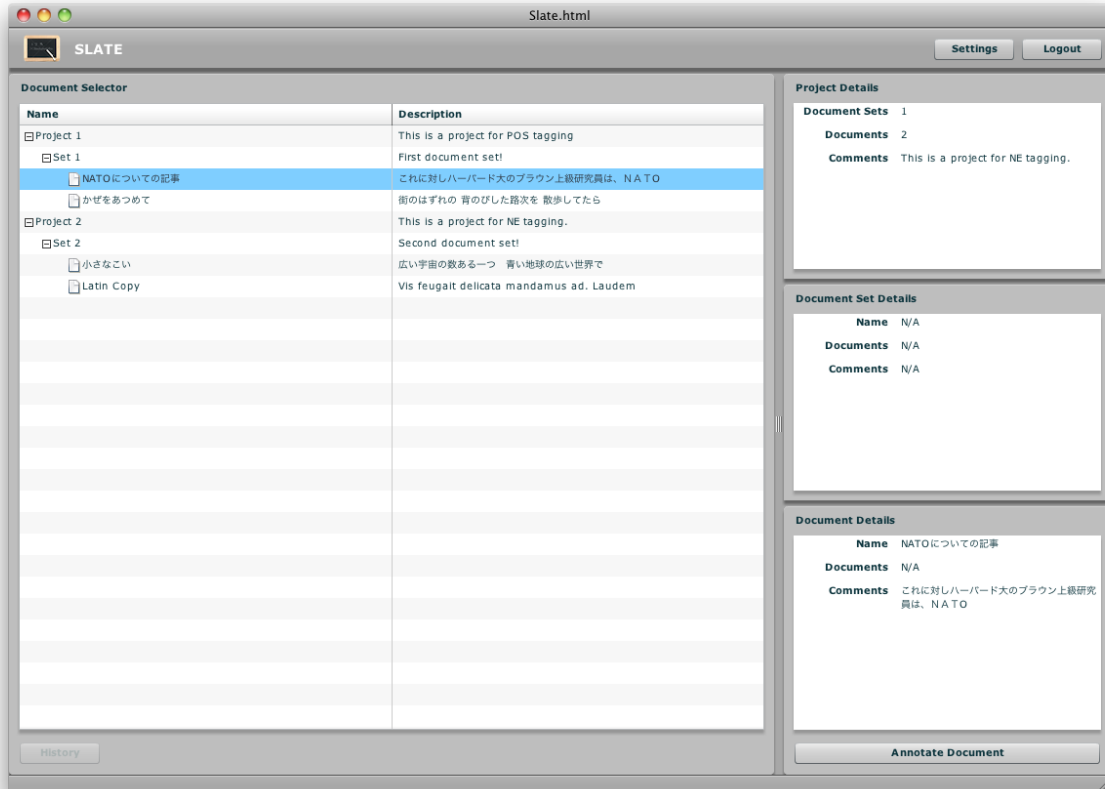
Acknowledgments

Figure 6: SLATE: Document Select Screen with multiple projects and multiple languages

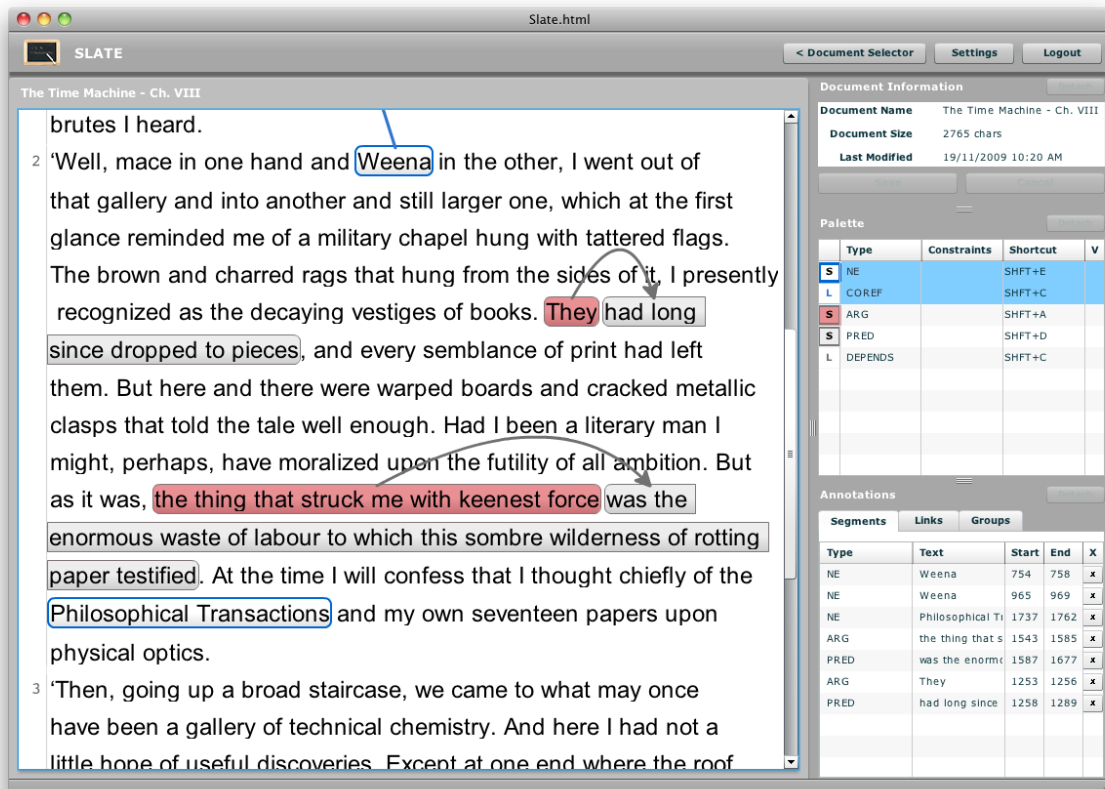
Figure 7: SLATE: Annotation Screen with English text showing coreference-chains and predicate-argument dependency

References

Constantin Or Asan and Constantin Orăsan. 2003. Palinka: A highly customisable tool for discourse annotation. In *Proc. of the 4 th SIGdial Workshop on Discourse and Dialogue, ACL '03*, pages 39–43.

Steven Bird, David Day, John S. Garofolo, John Henderson, Christophe Laprun, and Mark Liberman. 2000. Atlas: A flexible and extensible architecture for linguistic annotation. *CoRR*, cs.CL/0007022.

Callisto. 2002. http://callisto.mitre.org.

Eugene Charniak. 1993. *Statistical Language Learning*. The MIT Press.

H. Cunningham, D. Maynard, K. Bontcheva, and V. Tablan. 2002. Gate: A framework and graphical development environment for robust nlp tools and applications. In *Proc. of the 40th Annual Meeting of the ACL.*

Mark Davies. 2009. Contemporary American English (1990-2008+). *International Journal of Corpus Linguistics*, 14(2):159–190.

Glynn Dennis, Brad Sherman, Douglas Hosack, Jun Yang, Wei Gao, H. Clifford Lane, and Richard Lempicki. 2003. David: Database for annotation, visualization, and integrated discovery. *Genome Biology*, 4(5):P3+.

Stefanie Dipper, Michael Götze, and Manfred Stede. 2004. Simple annotation tools for complex annotation tasks: An evaluation. In *Proc. of the LREC Workshop on XML-based Richly Annotated Corpora.*

Ryu Iida, Mamoru Komachi, Kentaro Inui, and Yuji Matsumoto. 2007. Annotating a japanese text corpus with predicate-argument and coreference relations. In *Proc. of the Linguistic Annotation Workshop*, pages 132–139, Prague, Czech Republic, June. Association for Computational Linguistics.

Paul Kingsbury and Martha Palmer. 2002. From Treebank to PropBank. In *Proc. of the 3rd International Conference on Language Resources and Evaluation (LREC 2002)*, pages 1989–1993.

Christopher D. Manning and Hinrich Schuetze. 1999. *Foundations of Statistical Natural Language Processing*. The MIT Press, 1 edition, June.

Mitchell P. Marcus, Beatrice Santorini, and Mary Ann Marcinkiewicz. 1993. Building a large annotated corpus of English: The Penn Treebank. *Computational Linguistics*, 19(2):313–330.

Eleni Miltsakaki, Rashmi Prasad, Aravind Joshi, and Bonnie Webber. 2004. The penn discourse treebank. In *Proc. of LREC 2004*.

Christoph Mueller and Michael Strube. 2001. MMAX: A tool for the annotation of multi-modal corpora. In *Proc. of the 2nd IJCAI Workshop on Knowledge and Reasoning in Practical Dialogue Systems*, pages 45–50.

Masaki Noguchi, Kenta Miyoshi, Takenobu Tokunaga, Ryu Iida, Mamoru Komachi, and Kentaro Inui. 2008. Multiple purpose annotation using SLAT – segment and link-based annotation tool. *Proc. of 2nd Linguistic Annotation Workshop*, pages 61–64.

B. C. Russell, A. Torralba, K. P. Murphy, and W. T. Freeman. 2005. Labelme: A database and web-based tool for image annotation. Technical report, Tech. Rep. MIT-CSAIL-TR-2005-056, Massachusetts Institute of Technology.

Maik Stührenberg, Daniela Goecke, Nils Diewald, Alexander Mehler, and Irene Cramer. 2007. Web-based annotation of anaphoric relations and lexical chains. In *Proc. of the Linguistic Annotation Workshop*, pages 140–147, Prague, Czech Republic, June. Association for Computational Linguistics.

Tetsuro Takahashi and Kentaro Inui. 2006. A multipurpose corpus annotation tool: Tagrin. *Proc. of the 12th Annual Conference on Natural Language Processing*, pages 228–231.