Annotation Trees: LDC's customizable, extensible, scalable annotation infrastructure

Jonathan Wright, Kira Griffit, Joe Ellis, Stephanie Strassel, Brendan Callahan

Linguistic Data Consortium
University of Pennsylvania, Philadelphia, PA
{jdwright,kiragrif,joellis,strassel,bcal}@ldc.upenn.edu

Abstract

In recent months, LDC has developed a web-based annotation infrastructure centered around a tree model of annotations and a Ruby on Rails application called the LDC User Interface (LUI). The effort aims to centralize all annotation into this single platform, which means annotation is always available remotely, with no more software required than a web browser. While the design is monolithic in the sense of handling any number of annotation projects, it is also scalable, as it is distributed over many physical and virtual machines. Furthermore, minimizing customization was a core design principle, and new functionality can be plugged in without writing a full application. The creation and customization of GUIs is itself done through the web interface, without writing code, with the aim of eventually allowing project managers to create a new task without developer intervention. Many of the desirable features follow from the model of annotations as trees, and the operationalization of annotation as tree modification.

Keywords: annotation, web, software

1. Introduction

Under the auspices of three DARPA natural language programs, LDC has developed a new web-based annotation infrastructure that goes beyond typical notions of annotation tools. The central component of the framework is the generalized model of annotation data as a tree of types, hence the idea of "Annotation Trees". A variety of other design elements were built around this concept to address LDC's diverse annotation requirements, with the goal of supporting all annotation projects through a single platform. The primary piece of software is a Ruby on Rails application called the LDC User Interface (LUI). The current purpose of LUI is to support annotation efforts for LDC sponsored projects, not only through annotation itself, but related tasks like reporting. Here we discuss the current state of affairs, roughly 6 months into development, as well as future plans.

2. History

As part of DARPA's Machine Reading Program (MR), where multiple domains and tasks was part of the program design, LDC developed technical infrastructure for versatile and generalized annotation. The challenges of MR, namely the rapid development of multiple domains, represented a microcosm of LDC annotation project operations, since LDC resources are highly distributed across projects. In other words, while other projects may themselves be less varied compared to MR, LDC staff typically support multiple annotation projects, creating similar time constraints on individuals. In practice this often leads to reinventing the wheel; there is often no time to do otherwise. The design of MR brought this problem into focus, and the annotation infrastructure developed was designed from the beginning to be maximally reusable, which amounts to easily customizable. Early reports on this work are Strassel et al. (2010) and Maeda et al. (2010). However, the infrastructure as of mid 2011 was critically limited by a desktop oriented, rather than web oriented, design. During the MR Phase 3 evaluation of 2011, where remote use was mandatory, a new Ruby on Rails application was developed using the existing Annotation Tree model. During early 2012, the application has grown to support several projects, but primarily development has supported DARPA's BOLT and RATS projects.

3. Motivation

3.1. Logistical

Increasingly, remote annotation is either desired or absolutely necessary, since language experts can not always be found among in house staff. Furthermore, much annotation work occurs on nights and weekends when LDC is not open. Desktop applications are impractical due to problems of installation, annotator tracking, data transfer, etc. Cross platform software compatibility and the distribution of software has continually been a problem in cases where desktop software was used. Concurrent annotation is another concern, even when annotation is done in the office, and an efficient means of integrating the work of large teams is needed.

3.2. Technical

As annotation projects change, supporting code naturally changes as well. A particular change in the annotation targets or guidelines must be reflected in code that governs the user interface, and likely code that produces or formats the final product that forms the corpus. At a minimum, this involves changing a string somewhere. At the other extreme, a new project or new domain may be so different that no code can truly be reused. Add to this the problem of database reuse. The schema for a database changes from project to project, and even slight changes typically require brand new databases. The problem then is to maximally separate those parts of the infrastructure that might be changed from those that won't.

4. Infrastructure Design

4.1. User Centric Access

The front end for the new infrastructure is a Ruby on Rails application called the LDC User Interface or LUI, since it centers around the secure authentication and authorization of each visitor to the web site(s). Following the popular online Rails tutorial by Michael Hartl[1], the application began as nothing more than a site where a user could log in and visit their own (basically empty) profile. However, this user tracking kernel is an important design element around which everything, projects, tasks, workflows, annotation tools, etc., are built. Almost every URL that points to LUI results in user authentication, which ensures users access the projects and tools intended by project managers. Furthermore, careless or malicious access to data is not possible.

4.2. Monolithic Interface

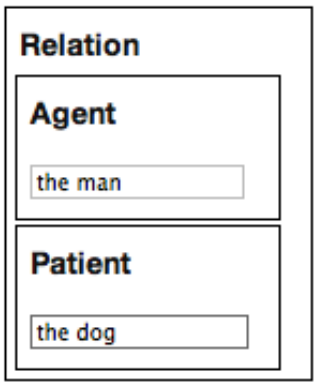
LUI is designed as a single interface for all annotation projects. A user can navigate among all projects and utilize all tools, assuming they have permission to do so. The benefits to project managers and annotators are obvious: there's a single point of entry and a unified user experience, where possible. While annotating named entities in text and performing speaker ID auditing of phone calls will be completely different, the same user can manager or receive assignments for these tasks with the same interface. The obvious potential drawbacks of a monolithic design are

1. the development difficulty of an all-in-one tool, and
2. the scalability of such a tool.

Nevertheless, the benefit of the monolithic design led to solutions to these concerns, details of which are discussed in following sections. Regarding (1), the notion of "tool" in this framework is somewhat misleading, and the infrastructure is designed such that a new tool is basically a new web page that plugs in to existing functionality. Regarding (2), LUI is deployed on multiple virtual machines with separate URLs; each instance is identical and points to the same databases. In addition, while application data like users and projects are in a central database, annotation data is stored across multiple physical or virtual machines. In short, the infrastructure scales by the addition of servers to the cluster.

4.3. Annotation Trees

To address reuse and customization, annotations were modeled with classes that could be instantiated, following the Object Oriented Programming paradigm. A domain specific language (DSL) for class definitions was created to configure the desktop application used during MR, such that the annotation task was defined at runtime by the DSL, outside the program code. Since the DSL classes pointed to other classes, the instantiation of a Root class resulted in a tree structure, hence "Annotation Trees". Annotation Trees capitalize on the typical convergence of three factors: annotation targets, GUI design (i.e. widget layout), and output structure. For example, if you want to annotate relations in text, along with their agents and patients, this forms a conceptual tree structure, where each relation has two subparts, the agent and patient. Since the layouts of GUIs are tree structured, and data formats like XML and JSON are tree structured, you likely have an isomorphism for all three structures. Enforcing this isomorphism, even when not ideal, greatly simplifies development and customization. A toy example might be the following:




```
{
 label: {
  value: "relation"
  },
 agent: {
  label: {
   value: "agent";
  },
  text: {
   value: {
    docid: "doc1",
    beg: 10,
    end: 16
   }
 },
 patient: {
  label: {
   value: "patient"
  },
  text: {
   value: {
    docid: "doc2",
    beg: 123,
    end: 129
   }
  }
 }
}
```

Figure 1: Simple GUI layout and its isomorphic data tree.

More realistic examples can be seen in the screen shots at the end of the paper. Eventually the domain specific language was abandoned in favor of a graphical editor that used the same concept (see Section 4.5.1). A user can create new widget types, set their CSS style, indicate their par-

[1] ruby.railstutorial.org

ent/child relationships, and see the resulting GUI layout in a read only mode. When the GUI is used in a live task, it will save data in an isomorphic fashion without any configuration required (see Fig 1. and the following section on data storage).

4.4. Data Storage

The framework has a mixed database backend. The application data, meaning things like users, projects, tasks, assignments, etc., are stored in a MySQL[2] database, while the annotation data is stored in a MongoDB[3] database. MongoDB is among several new "schema-less" databases; specifically we can say that MongoDB is a document store, rather than a relational database. The "documents" of a MongoDB instance are JSON objects, like the JSON object shown in Fig. 1. This is a natural storage mechanism for annotation trees. While there are ways to store trees in relational databases, MongoDB is designed to do so efficiently. Dynamic data, in other words, the annotations themselves, are stored this way in the database. The web application reads such a tree and "elaborates" it, adding the static information like labels and CSS styling, and creates a recursive `<div>` structure in HTML for display in a browser.

4.5. Minimizing Customization

There is a common programming paradigm known as Model-View-Controller or MVC, where an application is separated into the GUI (view), the data (model), and some logic for populating the GUI with the data (controller). In web applications, the view is the webpage, and the model roughly equates to the database design. A core design principle of LUI was to minimize the amount of customization necessary, and this is best viewed through the MVC paradigm. In short, customization of LUI to a new task is focused on the views, while no customization of the models or database design is necessary at all (specifically, no customization of the annotation database).

4.5.1. Namespace Editor

The namespace editor allows users to create task specific layouts for the annotation framework (Fig. 4). When a new namespace is created in the system, a new tree containing only a Root node is created. The control section on the right side of the editor allows users to modify the tree structure. As the user makes changes within the control section, the workspace view on the left side of the screen automatically updates with the current representation of the tree as it appears in the HTML DOM. The user must click the add widget link at the top of the controls section to create a new widget, and a dialog prompts them to specify a widget type and name. To edit existing widgets, the user clicks on the widget's name in the control section to expand the widget specific editor. The children of branching nodes can be rearranged using a simple drag and drop interface. The editor also offers a full screen preview of each layout, which allows any layout created within the editor to double as a mockup.

[2] www.mysql.com

[3] www.mongodb.org

4.5.2. Style Manager

Within the namespace editor, a style profile is created and mapped to every new widget. The style profile of any widget can be edited via the style manager, which displays an editable list of existing style profiles within (Fig. 5). Within the style manager, users are able to assign CSS attributes to any style profile. As a user makes changes to style profiles assigned to widgets in the workspace, they will see those changes update automatically in the workspace. The automatic page updating is accomplished via AJAX, letting the user create entire namespaces without having to refresh the page. Finally, these attributes are not stored with the annotation data, but attached to the trees on every web request, thus the attributes can be safely modified during the course of annotation.

4.5.3. Javascript Logic

Much of the customization of a new task can be isolated to the JavaScript loaded with a web page, for example, cosmetics, custom verbiage, and the activation/deactivation of input elements as an annotator enter data. While a general purpose JavaScript file is loaded for all annotation tasks, this general file expects the existence of a per-task file as well. The general purpose file controls the common behavior across all layouts, including the saving of actions taken by annotators, the creation of complex widgets that require JavaScript, the setting of listeners which occur on user actions, etc. The task specific file adds additional behavior without disrupting the general purpose behavior (unless that's the intent of course). For example, when a user tries to mark an assignment as complete, which is handled in the general purpose file, there is a hook for the task specific file to confirm that the user has completed all the task-specific steps required of them, and block the submission to the server if necessary.

4.5.4. Plug-in Workflows

One of the challenging aspects of annotation projects is workflow management, particular if one wants to maximize generality and minimize customization. Ideally, a new annotation task can utilize an existing workflow, e.g. given a set of assignments, randomly assign them as annotators log in. When this is not possible, and a developer must create a new workflow, LUI allows a new piece of logic to be dropped into place. In LUI, each workflow is itself a model, a Ruby class that wraps a Workflows table in the application database. Each workflow is essentially a state machine that moves users and data through a series of states, depending on the messages it receives from the application/user. Therefore to create a custom workflow, the developer creates a single Ruby class in the workflows directory which implements this state transition logic. The rest of the application passes messages to the workflow which then updates states accordingly, but the application requires no changes, other than this new workflow class.

4.6. Annotation Operations

Since all annotation data is modeled as generalized trees, annotation itself is modeled as a sequence of tree operations. The most important operations are as follows:

create-nodes When annotation begins, an empty tree is initialized to represent the empty annotation GUI presented to the user. This operation is called recursively, beginning with the root type, to produce the tree. It is also called after add-child, see below.

add-child Annotation GUIs are mostly static in shape, and dynamic needs are mainly met through a single List widget type. Where lists of annotations are necessary, a List widget is placed in the GUI. The add-child operation adds a new node to this list, followed by create-nodes to fill out the required subtree.

delete-node This operation deletes part of a tree, typically from a List widget.

change-value Most annotation is implemented with this operation. The leaves in the annotation tree are analogues to the columns of a relational database table, bearing the actual annotation data (judgement, comment, label, etc.). Differing views (radiobuttons, text boxes, etc.) perform alike, i.e. they change the value stored in a leaf node.

Complex operations can be composed from simple ones. For example, a button can be configured to send a sequence of operations as a single message to the server. During annotation, the full data structure visible to the annotator is not being sent back and forth from browser to server, only the changes are communicated.

4.7. Logging

The above operationalization of annotation, and the logging of these operations, allows for the following features.

real time updates Each action the annotator takes is immediately sent to the server to update the data, so an explicit "save" is never necessary. Ajax (Asynchronous Javascript) is used to avoid refreshing the page, which would be disruptive to annotation.

offline annotation New features of HTML5 may allow for annotation to proceed during a connectivity interruption. The operations would be queued locally for future syncing with the server.

accurate tracking Timestamps are logged with every operation, so fine grained tracking of annotators is possible.

version control Since the current state of the annotation tree is due to the series of operations that has been logged, previous versions of the data can be reconstructed.

5. Functionality

5.1. Basic Data Entry

Much annotation amounts filling out a complex HTML form of radio buttons, text boxes, etc. The key feature of LUI here is the instantaneous saving of every meaningful change, eliminating the need for a "submit" or "save" feature. Here "submitting" really amounts to moving on to the next assignment.

5.2. Text Annotation

Text annotation, in the sense of capturing strings from text as well as a pointer into the text (stand-off annotation), can be complex for many tasks and challenging to implement. LUI follows the model used in the MR desktop tool, where arbitrary strings in a document pane can be selected and captured into special text boxes in the annotation pane. When such an annotation is made, the selected text is underlined, and the (abbreviated) text is pasted into the text box within the annotation pane. As the user continues annotation of a document, navigation is possible by clicking on the text boxes, or on the underlines, to cause scrolling in the other pane. Multiple, overlapping underlines are possible, which is often the case in text annotation tasks.

5.3. Audio Capability

Audio playback is provided by a javascript plugin called JPlayer[4], which wraps the HTML5 capability of modern browsers. Currently only simple playback is provided, but future development will focus on waveform display, timestamped annotations, etc. JPlayer also provides video playback, another future direction for LUI. HTML5 features are key in obviating any special installation on the users' part, other than having a modern web browser.

5.4. Coreference

For text based annotation, coreference is increasingly important, and has been a central feature to the framework from early on. A drag and drop interface was designed where annotators drag mentions (or use the appropriate key strokes) from the left hand column to the right hand column, arranging them into boxes that represent entities (Fig. 4). This model has been extended to Speaker ID auditing, where instead of text, mentions have audio players embedded and annotators listen and group the mentions into entities based on what they hear.

5.5. Search

Some servers in the cluster run the Solr[5] search engine to provide text search capabilities to users.

5.6. Workflow Management

Workflows can be assigned to tasks via the interface, and managers can track progress by viewing the states of users and assignments. For example, does a particular user currently have an assignment, or how many assignments have been completed so far.

5.7. Reporting

An important corollary to annotation is reporting on that annotation, to managers, external sponsors, even the annotators themselves. The strong authentication backbone of LUI makes customized reporting easy, and a customizable reporting mechanism exists, similar to the other plug and play features of the annotation infrastructure. When a Ruby file that executes the necessary database query is added to the codebase, users have access to a new report that can be executed on demand.

[4] www.jplayer.org

[5] lucene.apache.org/solr/

6. Future Directions

6.1. Output Transformations

An important part of the MR desktop software was its ability to perform structural transformations on the annotation trees. In MR, annotation structure and the ontological structure of the final data product were typically not in alignment. The domain specific language used to define the annotation tree structure had a notation for rearranging elements, allowing for easy output creation without changing the native format of the annotations. A similar feature will be added to the current namespace (see Section 4.5.1). The RESTful interface of LUI can then be queried over HTTP to receive annotation data in various forms that may not be identical to the stored objects. Currently such data requests are handled by custom code in the application.

6.2. Multimedia

So far, the use of audio in LUI has been limited to auditing, i.e. making judgements on the played audio segments. However, the eventual goal is fully functional audio annotation, transcription, even acoustic analysis, tasks normally performed with desktop applications. Furthermore, the JPlayer plugin, already in use for audio, also plays video, so video annotation is in the development path as well.

6.3. Treebanking

The tree based storage, display, and annotation operationalization makes treebanking an obvious direction for development. The key here will be providing the necessary key and mouse bindings to provide annotators with an interface similar to the software they currently use.

6.4. Annotation as a Service

Following the model of software as a service, the possibility emerges of providing annotation as a service. The basic service already exists: remote annotation without software installation. What remains is allowing the user to determine the data and the task, and granting the authorization to download the data that they create. Assuming the appropriate permissions and resources can be established, the technical functionality will already be in place.

7. Conclusion

The LUI infrastructure and annotation tree model will provide the next generation of annotation "tools" for the LDC, where a "tool" is now just a customized configuration of existing components on a web page. In an environment where there is no real distinction between rapid prototyping and rapid development of the final product, this infrastructure promises to improve annotation efficiency. Furthermore, it lays the groundwork for providing an annotation and/or research service outside of our sponsored projects. Those interested in a demonstration should contact the LDC so that appropriate access can be granted. While anyone can visit one of the LUI URLs and create an account, a new user won't have sufficient authorization to access any tools or data.

8. Acknowledgements

This material is based upon work supported by the Defense Advanced Research Projects Agency (DARPA) under Contract No. D10PC20016. Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the Defense Advanced Research Projects Agency or its Contracting Agent, the U.S. Department of the Interior, National Business Center, Acquisition Services Directorate.

BOLT: This material is based upon work supported by the Defense Advanced Research Projects Agency (DARPA) under Contract No. HR0011-11-C-0145. The content of this paper does not necessarily reflect the position or the policy of the Government, and no official endorsement should be inferred.

9. References

Kazuaki Maeda, Haejoong Lee, Stephen Grimes, Jonathan Wright, Robert Parker, David Lee, and Andrea Mazzucchi. 2010. Technical infrastructure at linguistic data consortium: Software and hardware resources for linguistic data creation. In Nicoletta Calzolari (Conference Chair), Khalid Choukri, Bente Maegaard, Joseph Mariani, Jan Odijk, Stelios Piperidis, Mike Rosner, and Daniel Tapias, editors, *Proceedings of the Seventh conference on International Language Resources and Evaluation (LREC'10)*, Valletta, Malta, may. European Language Resources Association (ELRA).

Stephanie Strassel, Dan Adams, Henry Goldberg, Jonathan Herr, Ron Keesing, Daniel Oblinger, Heather Simpson, Robert Schrag, and Jonathan Wright. 2010. The darpa machine reading program - encouraging linguistic and reasoning research with a series of reading tasks. In Nicoletta Calzolari (Conference Chair), Khalid Choukri, Bente Maegaard, Joseph Mariani, Jan Odijk, Stelios Piperidis, Mike Rosner, and Daniel Tapias, editors, *Proceedings of the Seventh conference on International Language Resources and Evaluation (LREC'10)*, Valletta, Malta, may. European Language Resources Association (ELRA).

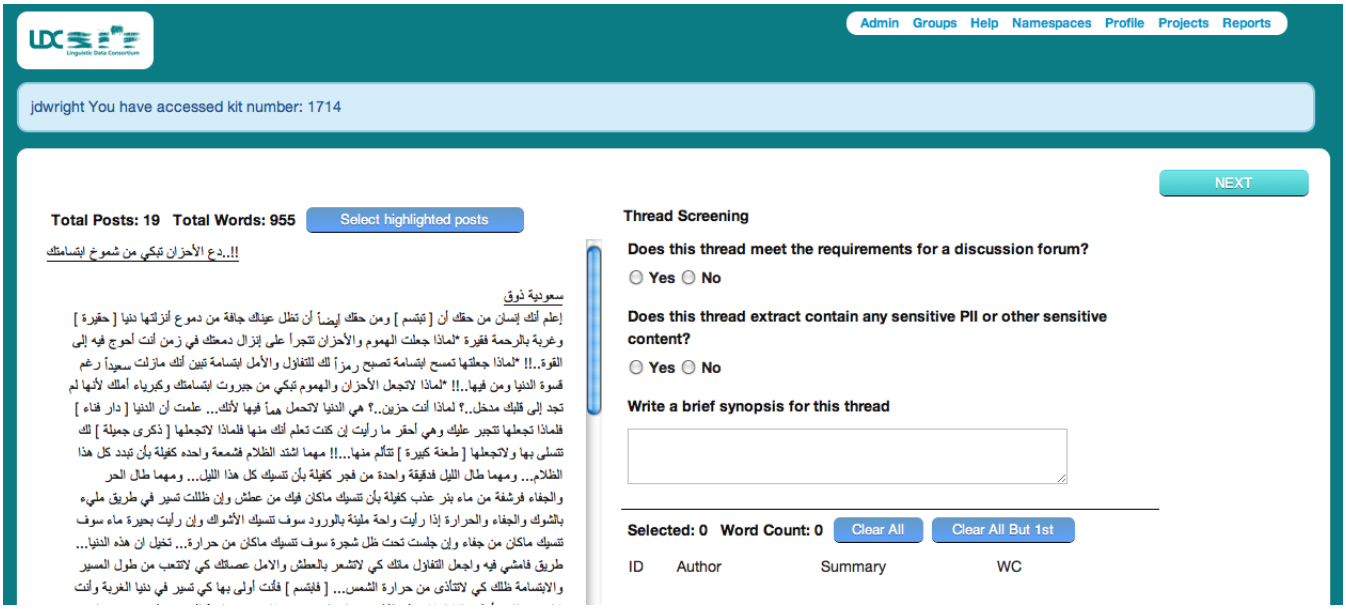


Figure 2: A snapshot of the BOLT triage tool.

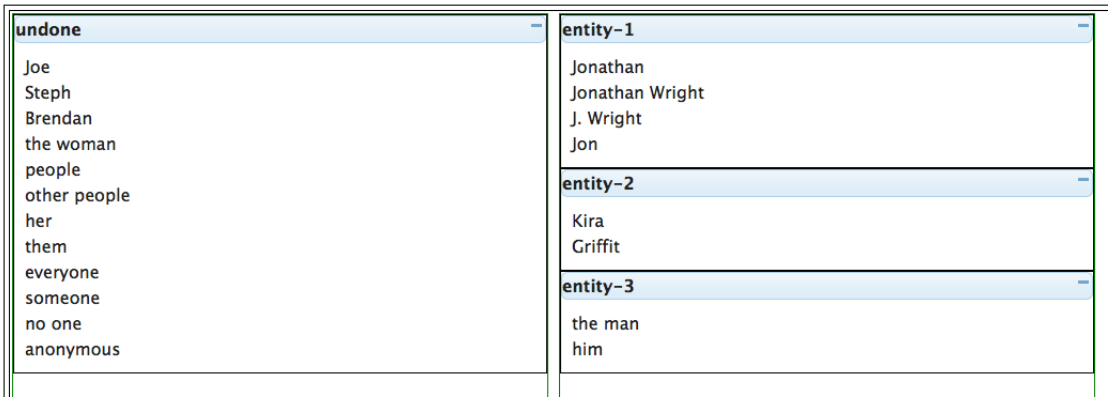


Figure 3: A snapshot of the coreference widget, with toy data.

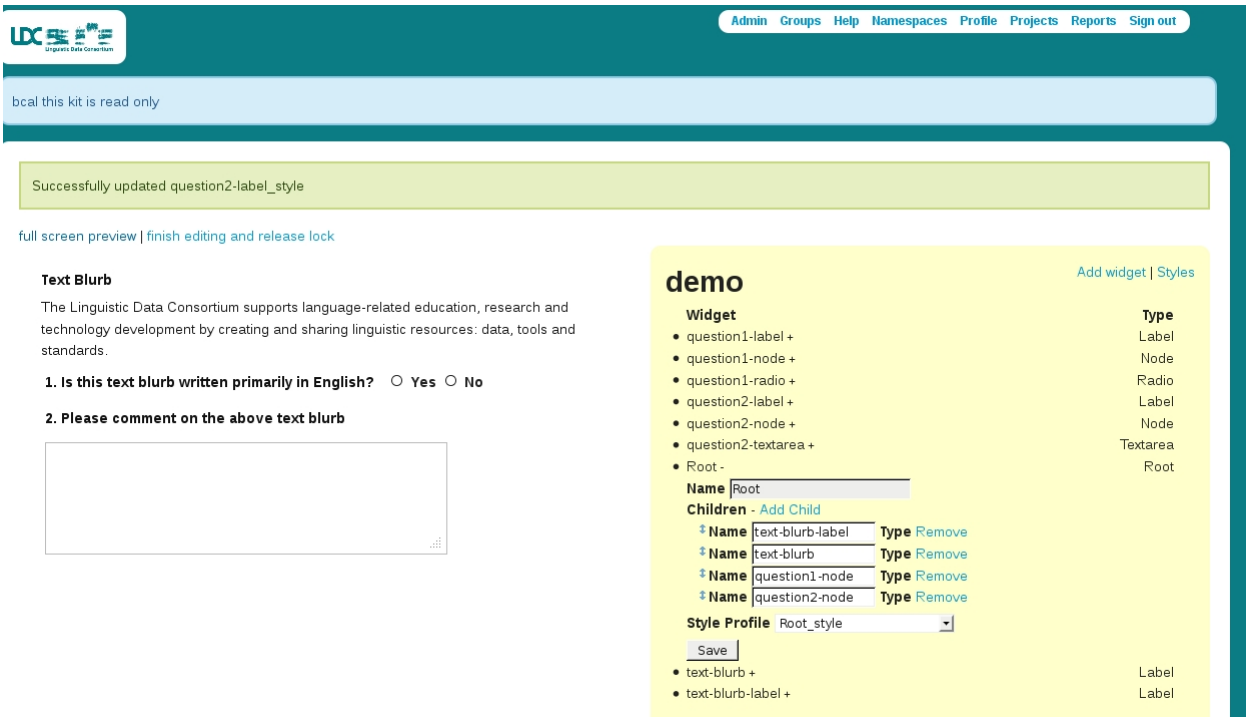


Figure 4: The namespace editor with a simple GUI layout.

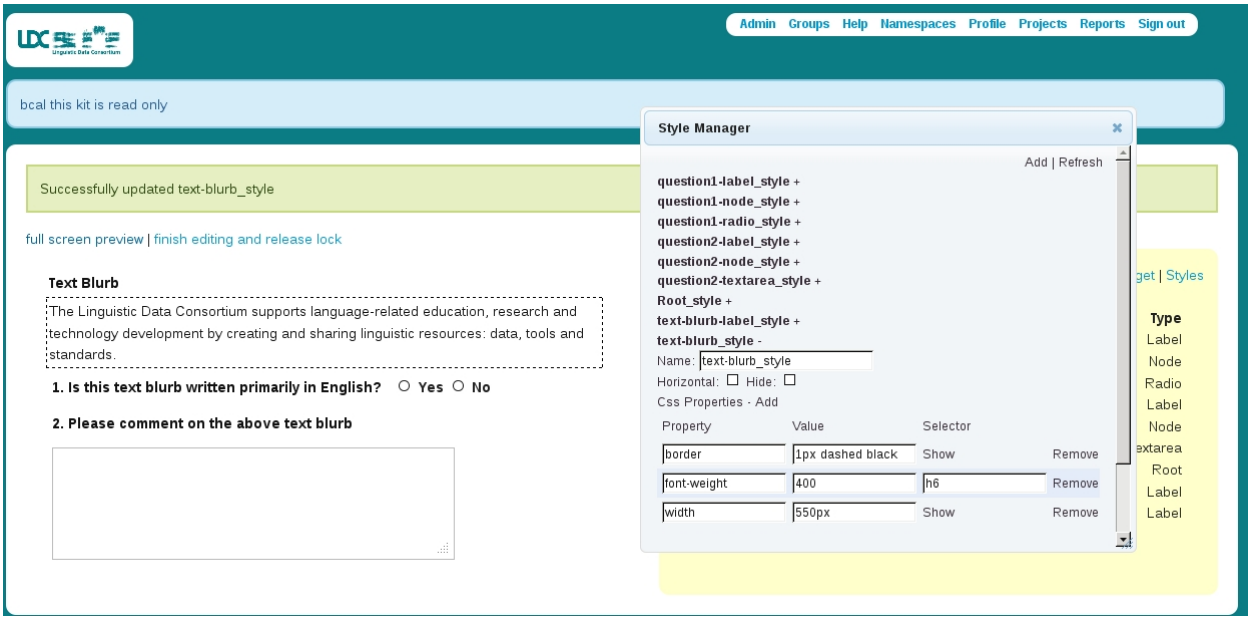


Figure 5: The style manager opened on the same GUI.