Towards Reusable Explanation Services in Protege

Yevgeny Kazakov[1], Pavel Klinov[2], and Alexander Stupnikov[3]

[1] The University of Ulm, Germany
yevgeny.kazakov@uni-ulm.de
[2] Stardog Union, USA
pavel@stardog.com
[3] Tyumen State University, Russia
saarus72@gmail.com

Abstract. We present several extensions of the explanation facility of the ontology editor Protege. Currently, explanations of OWL entailments in Protege are provided as justifications—minimal subsets of axioms that entail the given axiom. The plugin called 'explanation workbench' computes justifications using a black-box algorithm and displays them in a convenient way. Recently, several other (mostly glass-box) tools for computing justifications have been developed, and it would be of interest to use such tools in Protege. To facilitate the development of justification-based explanation plugins for Protege, we have separated the explanation workbench into two reusable components—a plugin for black-box computation of justifications and a plugin for displaying (any) justifications. Many glass-box methods compute justifications from proofs, and we have also developed a reusable plugin for this service that can be used with (any) proofs. In addition, we have developed an explanation plugin that displays such proofs directly. Both plugins can be used, e.g., with the proofs provided by the ELK reasoner. This paper describes design, features, and implementation of these plugins.

1 Introduction

Protégé is currently the *de facto* standard domain independent tool for working with OWL ontologies. It provides rich functionality for editing and browsing ontologies as well as integration with backend reasoning services [1]. The characteristic property of Protégé is its extensible and modular architecture which enables developers to easily add new features via pluggable modules (or *plugins* for short) which are automatically loaded at runtime using the OSGi framework. Most of user facing functionality such as reasoning, query answering, modularity, learning, etc. is implemented via plugins available from the Protégé Plugin Library.[4]

Importance of explanations for working with OWL ontologies has been long recognised and Protégé has a special extension point for plugins providing explanations services. It defines a high level API for getting explanations for entailments. Any explanation services plugin is responsible for two things: obtaining information about why an entailment holds and displaying that information to the user in a UI widget.

[4] https://protegewiki.stanford.edu/wiki/Protege_Plugin_Library

The Protégé distribution ships with one standard explanation plugin called "Explanations Workbench". It displays a particular kind of explanations called *justifications*, i.e. the minimal subsets of the ontology which are sufficient for the entailment to hold, and computes them using a black-box algorithm (cf. next section for distinction between black-box and glass-box algorithms). The plugin provides some useful insights into the structure of explanations, e.g. identifies axioms which occur in multiple, or even all, justifications.

As useful as it is, the standard plugin is hard to extend or reuse. The key issue is that it combines both presentation and computation aspects making it difficult to reuse either functionality. For example, if a particular reasoner provides efficient means for computing justifications, it would be desirable to use it together with the built-in presentation method. On the other hand, one may want to customise UI for a particular use case but reuse the computation method. Both such tasks currently require modification, rather than extension, of the explanation workbench. Our first goal is to rectify this by providing reusable and flexible explanation architecture.

Our second goal is to complement justification-based explanation services in Protégé with proof-based explanations. Proofs provide insights into which steps (inferences) are performed to derive a particular result from the ontology. Similarly to justifications we aim at providing reusable services which allow other developers to plug in their proof generators or customise proof rendering.

The remainder of the paper is structured as follows: Section 2 defines terminology and introduces different approaches to explanations. Section 3 presents our architecture of explanation services in Protégé. Section 4 describes the main features and optimisation for both justification-based and proof-based explanation services. Section 5 concludes the paper.

2 Explanations

We begin with terminology which will allow us to formally define the guarantees that our plugins provide to the user. As usual, an (OWL) *ontology* is a set of (OWL) *axioms*. Given an ontology $\mathcal{O}$ and an axiom α, a *justification* (for the entailment $\mathcal{O} \models \alpha$) is a minimal subset $J \subseteq \mathcal{O}$ such that $J \models \alpha$ [2].

An *inference* is an object inf of the form $\langle \alpha_1, \ldots, \alpha_n \vdash \alpha \rangle$ where $\alpha_1, \ldots, \alpha_n$ is a (possibly empty) sequence of axioms called *the premises* of the inference and α is an axiom that is the *conclusion* of the inference. Informally, an inference corresponds to application of a particular inference rule to particular axioms (asserted or inferred) to derive an axiom. Axioms asserted in the ontology can also appear as conclusions in inferences. An inference $\langle \alpha_1, \ldots, \alpha_n \vdash \alpha \rangle$ is *sound* if $\alpha_1, \ldots, \alpha_n \models \alpha$.

An *inference set* is any set of inferences. Let $\mathcal{O}$ be an ontology. A *derivation* (from $\mathcal{O}$) using an inference set I is a sequence of inferences $d = \langle \mathrm{inf}_1, \ldots, \mathrm{inf}_k \rangle$ from I ($k \geq 0$) such that for every i with $1 \leq i \leq k$, and each premise α of inf_i that does not appear in $\mathcal{O}$, there exists $j < i$ such that α is the conclusion of inf_j. We say that an axiom α is *derivable* from $\mathcal{O}$ using an inference set I if there exists a derivation d such that α is the conclusion of the last inference in d. In this set we will often call the

inference set I a *proof* for α. A proof I is *complete* for the entailment $\mathcal{O} \models \alpha$ if for each justification $J \subseteq \mathcal{O}$ for $\mathcal{O} \models \alpha$, α is derivable from J using I.

A *proof structure* over inferences in I is a triplet $P = \langle N, L, S \rangle$ where N is a set of nodes, L a labeling function that assigns an axiom to each $n \in N$, and S a set of *proof steps* of the form $\langle n_1, \ldots, n_k \vdash n \rangle$ with $n_1, \ldots, n_k, n \in N$ such that for $\alpha_i = L(n_i)$ $(1 \leq i \leq k)$ and $\alpha = L(n)$, we have $\langle \alpha_1, \ldots, \alpha_k \vdash \alpha \rangle \in I$. Similarly to inferences, for a proof step $s = \langle n_1, \ldots, n_k \vdash n \rangle$, the nodes $n_1, \ldots, n_k$ are called the *premises* of s and the node n is called the *conclusion* of s. Similar to derivations, a proof structure over I represents a particular strategy of applications of inferences in I aimed at deriving a particular axiom. But unlike derivations, this strategy is not linear and might contain non-derivable axioms in the labels, e.g., due to cycles. Specifically, the proof structure P is *cyclic* if the binary relation $E = \{\langle n_i, n \rangle \mid \langle n_1, \ldots, n_k \vdash n \rangle \in N, 1 \leq i \leq k\}$ is *cyclic*, and otherwise it is acyclic. The *derivable nodes* of P w.r.t. an ontology $\mathcal{O}$ is the smallest subset $D \subseteq N$ of nodes such that (i) if $n \in N$ is labeled by an axiom in $\mathcal{O}$ then $n \in D$, and (ii) if $\langle n_1, \ldots, n_k \vdash n \rangle \in S$ and $n_1, \ldots, n_k \in D$ then $n \in D$. A proof structure P is *entailment-complete* for a node $n \in N$ such that $\mathcal{O} \models \alpha = L(n)$ if for every justification J for $\mathcal{O} \models \alpha$, n is derivable w.r.t. J.

Justifications and proofs offer two orthogonal approaches to explaining entailments: the former point to minimal subsets of the ontology which are necessary for the entailment to hold while the latter show specific inference steps. Justifications have the additional property that they can be computed in a reasoner-agnostic way using a *black-box* algorithm [3, 2]. In other words, the reasoner does not need to implement any non-standard reasoning service and is used as an oracle for entailment queries. Alternatively both justifications and proofs can be computed using *glass-box* algorithms which get information from inside the reasoner. Glass-box justification algorithms have been designed based on tableau tracing for expressive DLs [3] and connections between the $\mathcal{EL}$ reasoning and propositional SAT [4]. For proofs, the reasoner must offer some insight into the inference steps it takes to derive the entailment, so all proof generation algorithms are glass-box. One such algorithm for $\mathcal{EL}$ has recently been developed for the ELK reasoner [5, 6]. While the proof- and justification-based approach are fundamentally different, there are important connections between their results, in particular, justifications can be computed from proof inferences, e.g., using SAT solvers [7–12].

Our proof explanations plugin computes and displays acyclic (entailment-complete) proofs from the provided (entailment-complete) inference sets. Our justification explanation plugin displays the provided justifications. Additionally, our proof justification plugin computes (all) justifications from the reasoner-provided (entailment-complete) inference sets. Together with the black-box justification plugin, it can be used as a service for the justification-based explanation plugin. The plugins have many options that, for example, may filter out superfluous inferences while guaranteeing completeness w.r.t. entailments and justifications.

3 The New Explanation Services for Protégé

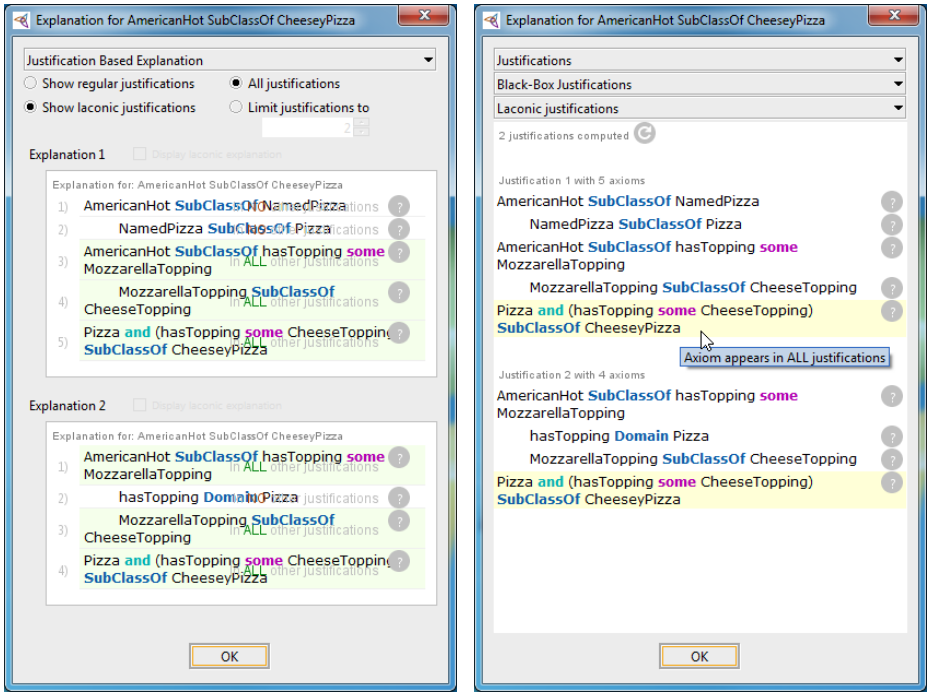
Protégé is a powerful ontology editor, which provides many possible ways of extending functionality and adding new features. This mechanism is based on so-called extension

points and plugins that can be provided using an OSGi framework or, specifically, its implementation used for the Eclipse IDE [13]. In a nutshell, *an extension point* is a Java abstract class or an interface which the *extending plugin* should extend or, respectively, implement in order to provide the additional functionality. For example, Protégé defines an extension point for *explanation services* using the code like in Listing 1.1.

Listing 1.1. Explanation service extension point of Protégé

```
abstract class ExplanationService {
  abstract JPanel explain(OWLAxiom axiom);
}
```

This means that any plugin that provides an explanation service should implement a function, that given an OWL axiom, draws a widget (JPanel) that somehow explains why this axiom was entailed. The *'explanation workbench'*[5] plugin [14] that is bundled with Protégé implements this method by first computing justifications for the axiom using a black-box procedure [2], and then displaying these justifications in a well-formatted way. An example output of this plugin is shown in Figure 1(a).



(a) (b)

Fig. 1. Comparison of the old (a) and the new (b) design showing laconic justifications [15]

[5] https://github.com/protegeproject/explanation-workbench

Unfortunately, it is not possible to use the explanation workbench for displaying explanations computed by other tools, e.g., those based on SAT solvers [7–12]. To address these limitations, we have separated the explanation workbench on two plugins: *'justification explanations'*[6]—a plugin for *displaying* justifications, and *'black-box justifications'*[7]—a plugin for *computing* justifications using a black-box procedure. These plugins are schematically shown in the left part of Figure 2. Here, within the 'justification explanations' plugin, we have defined another extension point called *justification service* using which other plugins can provide justifications that are then displayed by this plugin. The simplified code for defining this extension point is given in Listing 1.2.

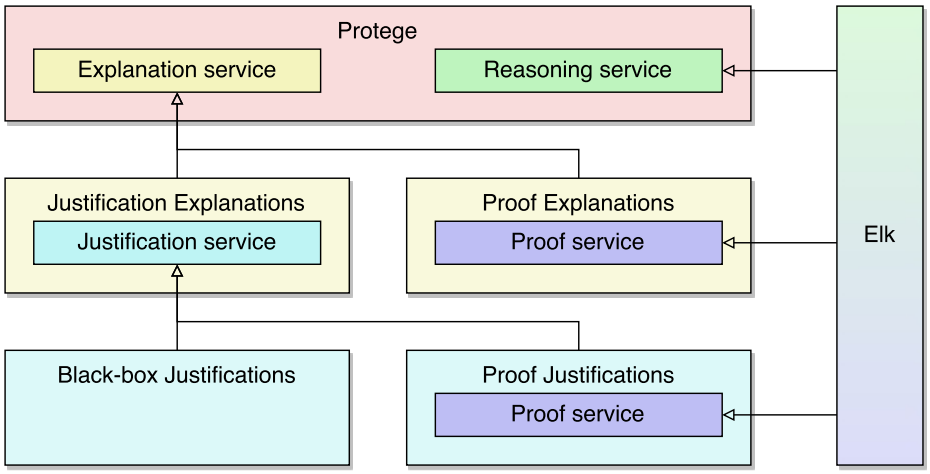


Fig. 2. A hierarchy of Protégé plugins for providing explanations

Listing 1.2. Justification service extension point of the 'justification explanations' plugin

```
abstract class JustificationService {
  abstract void enumerateJustifications(OWLAxiom
      entailment, JustificationListener listener);
}

interface JustificationListener {
  void foundJustification(Set<OWLAxiom> justification);
}
```

Based on this code, any plugin that provides a justification service, must implement the method `enumerateJustifications()` that, given an axiom, computes justifications for the entailment of this axiom and reports them one by one using the given

[6] https://github.com/liveontologies/protege-justification-explanation
[7] https://github.com/liveontologies/protege-black-box-justification

`JustificationListener` by calling the method `foundJustification()` with the computed justification as the argument. That is, the plugin can report each justification as soon as it is computed, without waiting for the method to return.

For example, the 'black-box justifications' plugin, shown in the bottom left part of Figure 2, implements this method by repeatedly calling the reasoner on subsets of the ontology to identify a minimal subset that entails the given axiom, reporting the found justification using the listener, and continuing searching for other minimal subsets.

Recently, a number of efficient *glass-box* methods for computing justifications have been developed, that use as an input an inference set, usually obtained by consequence-based procedures for $\mathcal{EL}$ ontologies [16–18]. Now, it is possible to use these tools with the re-engineered 'justification explanation' plugin. As an example, we have implemented a plugin *'proof justifications'*[8] shown in the bottom of Figure 2, that uses a resolution-based procedure for computing justifications from the Proof Utility Library (PULi).[9] In order to use arbitrary proofs for computing justifications, this plugin declares an extension point *proof service* with the simplified code shown in Listing 1.3.

Listing 1.3. Proof service extension point of the 'proof justifications' plugin

```
abstract class ProofService {
  abstract Proof<OWLAxiom> getProof(OWLAxiom entailment);
}

interface Proof<C> {
  Collection<? extends Inference<C>> getInferences(C
      conclusion);
}

interface Inference<C> {
  String getName();
  C getConclusion();
  List<? extends C> getPremises();
}
```

A plugin that provides a proof service, should implement a method `getProof()` that, given an axiom, returns a proof using which this axiom can be derived. The inferences of this proof can be recursively *unravelled* using the method `getInferences()` of `Proof`. The latest version of the ELK reasoner[10] provides an implementation of this extension point using a goal-directed tracing procedure for $\mathcal{EL}$ ontologies [5]. Note that, the inferences for each conclusion may be returned as any collection, e.g., a list or a set, but the premises of each inference should be a list since the order of premises is important to match inferences against the rules, e.g., for automated proof verification.

A proof provided by a proof service according to Listing 1.3 can be also regarded as an explanation for the entailment, and we next have developed a plugin *'proof ex-*

[8] https://github.com/liveontologies/protege-proof-justification
[9] https://github.com/liveontologies/puli
[10] https://github.com/liveontologies/elk-reasoner

planations'[11] that can display any such proof. This plugin is shown in the right part of Figure 2. Although the extension point *Proof service* of this plugin is similar to the extension point of the 'proof justifications' plugin, for technical reason, these two plugins cannot share a common extension point, so any plugin that implements a proof service must explicitly specify for which extension point it is provided.

As it is usual for OSGI plugins, all plugins shown in Figure 2 are *optional* and are not required for functionality of Protégé. In particular, if any other reasoner providing reasoning services but not proof services is used instead or in addition to ELK, it can be used for reasoning with ontologies as usual, but will not be used within the 'proof explanations' or 'proof justifications' plugins.

4 Features, Implementation, and Optimizations

In this section, we look more closely into design and implementation of the plugins shown in Figure 2.

Since, in general, there can be many plugins implementing each particular extension point, there should be a mechanism of selecting which plugin should be used. In case there are several plugins implementing the explanation service, Protégé displays a drop-down menu at the top of the explanation window in which the required explanation service can be chosen. The content of the window is then updated according to the selected plugin. See Figure 1(a) and Figure 1(b) comparing the windows for 'explanation workbench' (old design) and 'justification explanations' (new design) respectively.

We have adapted this principle for all other extension points. For example, if there are several plugins implementing the justification service, a drop-down menu listing these plugins appears. A similar design applies for plugin preferences. Each extension point in Figure 2, in addition to the corresponding methods in Listings 1.1–1.3, provides a method `getPreferences()` using which specific plugin settings can be displayed. The preferences for all installed plugins are then assembled in the Protégé preferences menu in the Explanations tab (see Figure 3). All plugins implementing a particular extension point, e.g., 'black-box justifications' and 'proof justifications', are listed in the respective 'General' tab of the plugin providing the extension point, and the specific settings for each plugin are located on a separate tab as shown in Figure 3.

We next consider the specific features of justification explanations and proof explanations in more detail.

4.1 Justification-based explanations

Most of the changes in the newly designed 'justification explanations' plugin were dictated by the improved performance of (glass-box) justification computation tools. Whereas the default 'explanation workbench' was usually able to compute only a few of justifications for real ontologies, the new tools can compute thousands of justifications in a few seconds (see Figure 4). Such large number of justifications is usually not possible to display in one window; even with less than a hundred of justifications there were some significant lags during both the opening and scrolling of the window.

[11] `https://github.com/liveontologies/protege-proof-explanation`

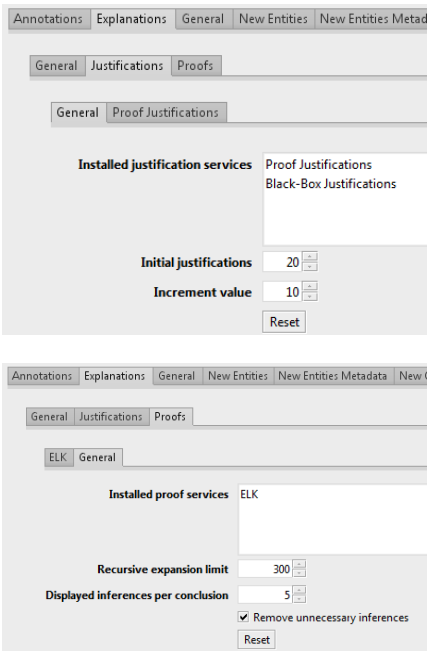


Fig. 3. Protégé explanation preferences

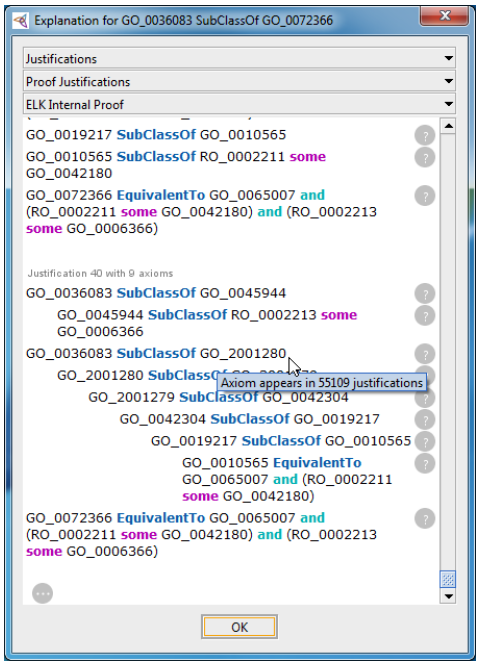


Fig. 4. Proof justifications

To improve the performance of the visual components, we have limited the number of justifications that can be (initially) displayed in the window. If the number of computed justification exceeds this limit, there is a possibility to load more justifications using a button in the bottom of the window (see Figure 4). Both, the the initial limit value and the maximal number of justifications that can be loaded by pressing the button, can be changed in the preferences of the plugin (see the upper part of Figure 3).

Even with the limited number of justifications, the plugin has exhibited a considerable delay before displaying the window. The delay was caused by the initial *sorting* of all computed justifications so that simpler justifications are shown first.[12] Sorting thousands of justifications seems unnecessary if only a few of them should be displayed. To address this problem, we store all computed justifications in a *priority queue* that allows retrieving elements in a specified (priority) order. Finding an element in the queue that is minimal with respect to the order is a logarithmic operation; it does not require all elements in the queue to be sorted.

We have made a few further changes to improve the overall 'look and feel' and make the components more native for Protégé. The original 'explanation workbench'

[12] The 'explanation workbench' prioritizes justifications first according to the number of different types of axiom constructors used, then according to the number of different types of concept constructors used, and, finally, according to the number of axioms in justifications—those with the lowest values in the lexicographical order appear first in the list.

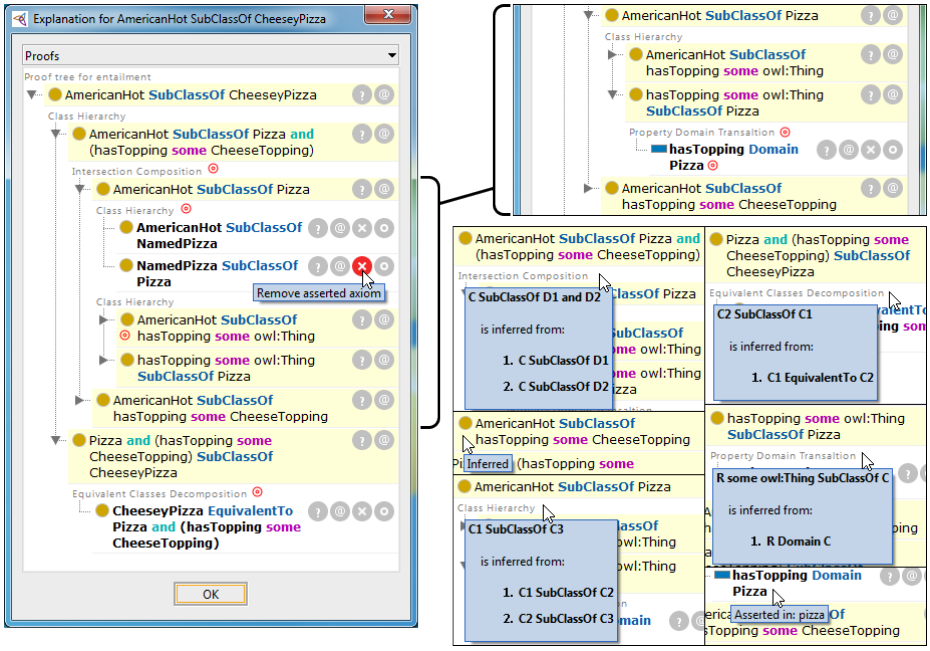


Fig. 5. The explanation window shown by the 'proof explanations' plugin

displays next to each axiom in justifications the number of justifications this axiom appears in. Since for large justifications this view becomes too crowded and also has problems with automated line breaking for long axioms (see the left part of Figure 1), we have moved this information into *tooltips*, shown when moving the mouse over the axiom (see Figure 4). We have also cached the values of the numbers; previously they were computed every time the window is redrawn. Finally, as in other Protégé components, we have highlighted in yellow the (entailed) axioms that are not present in the ontology. Such axioms appear, for example, when the 'show laconic justifications' setting of 'black-box justifications' is selected. In this case the axioms in justifications are 'simplified' to axioms that are not present in the ontology [15].

4.2 Proof-based explanations

The 'proof explanations' plugin is a new plugin that was designed from scratch. The main idea is to present to the user all relevant inferences that derive the axiom in question, and let the user explore these inferences interactively.

Naturally, such proofs can be presented in a tree-like component, where the nodes are labeled by axioms. Unfortunately, the standard Java component JTree has some limitations, which does not allow us to use it for displaying proofs. First, there is no easy way of displaying information about inferences since for this, a second type of nodes is needed. Second, wrapping node labels cannot be easily implemented. Third, the standard Protégé buttons for explaining, annotating, deleting, and editing axioms,

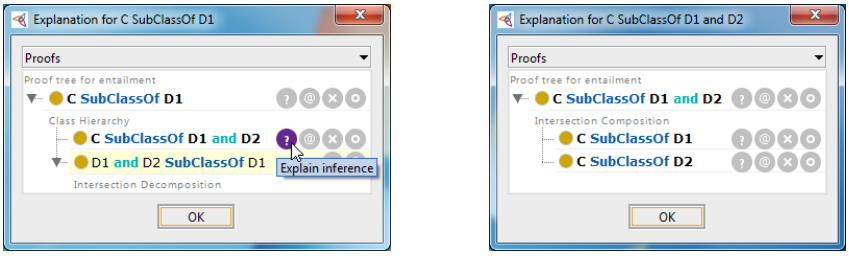


Fig. 6. Displaying proofs for cyclic inferences by the 'proof explanations' plugin

cannot be easily displayed in a JTree. Therefore, we chose to implement a custom tree-like component for proofs based on lists.[13] The result can be seen on the left of Figure 5.

In order to display this window, the inferences I obtained by the proof service (see Listing 1.3) are grouped in a proof structure $P = (N, L, S)$ as defined in Section 2, whose nodes N and labels $L(n)$ for $n \in N$ are respectively the nodes and the axiom labels of the proof tree. A straightforward solution would be to set N as the set of all axioms α appearing in I, define $L(\alpha) = \alpha$ for $\alpha \in N$, and set $S = I$. This proof, however, may be *cyclic*, which is undesirable since the user would not be able to completely expand the proof. This situation is rather common in practice. For example, $\mathcal{EL}$ procedures [16–18] often apply inferences that both *decompose* conjunctions: $\langle C \sqsubseteq D_1 \sqcap D_2 \vdash C \sqsubseteq D_1 \rangle$ and *compose* them: $\langle C \sqsubseteq D_1, C \sqsubseteq D_2 \vdash C \sqsubseteq D_1 \sqcap D_2 \rangle$. Figure 6 illustrates this situation: notice that the inference for $C \sqsubseteq D_1 \sqcap D_2$ cannot be expanded in the left proof, but can be expanded in the right proof.

To solve this problem we *eliminate cycles* in proofs. Intuitively, it is sufficient to consider only proof structures in which every branch does not contain duplicate labels. Specifically, if an inference producing a label of some node contains a premise that already appears on the path from the proof root, we do not display this inference. Removing such inferences from the proof tree, however, is not enough since this may result in some *dead nodes*—the nodes that cannot be derived (as defined in Section 2) due to the removed inferences. To determine if a node is dead (and thus should not be shown), we check if the label of the node is derivable using the inferences in I without using axioms on the path leading to this node. This can be easily verified in linear time (in the size of I) by simply removing from I all inferences whose conclusions are those axioms on the path, and checking if the label is derivable using the remaining inferences. It is easy to see that this operation also preserves *completeness* of proofs: if there is a (part of the) proof from some justification J of $\mathcal{O}$, it has an acyclic sub-proof. Arguably, completeness is a desirable property of proofs, since by expanding such proofs the user is able to find all possible subsets of axioms which entail the axiom of interest.

The 'proof explanations' plugin has a number of further interesting features. First, the proofs are dynamically updated after changes are made with axioms in the ontology. For example, when an axiom is deleted (see the left of Figure 5), the proof is automatically updated to remove the fragments that use this axiom (see the right of Figure 5).

[13] Specifically on MList used in Protégé for displaying lists of OWL objects

For axiom modifications, this may require a reasoner 'synchronization' to recompute the entailments (and hence, the inferences). To support the proof updates, the proof service must return a `DynamicProof`, which, in addition to the methods of `Proof` (see Listing 1.3), allows to register listeners to monitor for changes.

Next, the 'proof explanations' plugin provides several tooltips that display some additional useful information to the user. Apart from specifying if axioms are stated or inferred, tooltips can display examples for inferences used in the proofs. *Examples* are simple instances of inferences that can help the user to understand how the inference used in the proof is applied in general (see the right of Figure 5).

The plugin can be also customized using some preference settings (see the bottom of Figure 3). For example, the proof can be recursively expanded up to a certain limit of inferences (using ALT+click or long press), one can limit the number of inferences initially shown per conclusion, and some inferences can be automatically removed from the proof if this preserves completeness of the proof.

5 Conclusion

We have described a range of new Protégé plugins aimed at improving debugging experience for OWL ontologies. Some of these plugings, such as 'justification explanations' and 'black-box justifications' are obtained by re-engineering the existing 'explanation workbench' explanation plugin. Others, like 'proof explanations' and 'proof justifications' were built from scratch. The main goal was to being able to provide flexible and reusable explanation services for Protégé. This is achieved by separating computation of explanations from displaying explanations to the user.

There are many ways to further improve the described plugins. For example, the 'justification explanations' plugin currently does not support updates after changes like in the 'proof explanations' plugin. The 'proof explanations' and 'proof justifications' plugins can potentially reuse a common proof service, if it is moved to a new plugin that is added as common dependency of these two plugins. This would then allow other plugins to use proof services without additional extension points, just like it is possible now for ontology reasoners. However, currently Protégé lacks the mechanism to manage plugin dependencies, which poses some difficulties in this regards.

References

1. Musen, M.A.: The protégé project: a look back and a look forward. AI Matters **1**(4) (2015) 4–12
2. Kalyanpur, A., Parsia, B., Horridge, M., Sirin, E.: Finding all justifications of OWL DL entailments. In Aberer, K., Choi, K.S., Noy, N., Allemang, D., Lee, K.I., Nixon, L., Golbeck, J., Mika, P., Maynard, D., Mizoguchi, R., Schreiber, G., Cudré-Mauroux, P., eds.: Proc. 6th Int. Semantic Web Conf. (ISWC'07). Volume 4825 of LNCS., Springer (2007) 267–280
3. Kalyanpur, A.: Debugging and Repair of OWL Ontologies. PhD thesis, University of Maryland College Park, USA (2006)
4. Baader, F., Peñaloza, R., Suntisrivaraporn, B.: Pinpointing in the description logic EL^+. In Hertzberg, J., Beetz, M., Englert, R., eds.: KI 2007: Advances in Artificial Intelligence, 30th

Annual German Conference on AI, KI 2007, Osnabrück, Germany, September 10-13, 2007, Proceedings. Volume 4667 of Lecture Notes in Computer Science., Springer (2007) 52–67
5. Kazakov, Y., Klinov, P.: Goal-directed tracing of inferences in $\mathcal{EL}$ ontologies. In: The Semantic Web - ISWC 2014 - 13th International Semantic Web Conference, Riva del Garda, Italy, October 19-23, 2014. Proceedings, Part II. (2014) 196–211
6. Kazakov, Y., Klinov, P.: Advancing ELK: not only performance matters. In Calvanese, D., Konev, B., eds.: Proceedings of the 28th International Workshop on Description Logics, Athens,Greece, June 7-10, 2015. Volume 1350 of CEUR Workshop Proceedings., CEUR-WS.org (2015)
7. Sebastiani, R., Vescovi, M.: Axiom pinpointing in lightweight description logics via horn-sat encoding and conflict analysis. In Schmidt, R.A., ed.: Proc. 22st Conf. on Automated Deduction (CADE'09). Volume 5663 of Lecture Notes in Computer Science., Springer (2009) 84–99
8. Vescovi, M.: Exploiting SAT and SMT Techniques for Automated Reasoning and Ontology Manipulation in Description Logics. PhD thesis, University of Trento, Italy (2011)
9. Arif, M.F., Mencía, C., Marques-Silva, J.: Efficient axiom pinpointing with EL2MCS. In Hölldobler, S., Krötzsch, M., Peñaloza, R., Rudolph, S., eds.: Proc. 38th Annual German-Conf. on Artificial Intelligence (KI'15). Volume 9324 of Lecture Notes in Computer Science., Springer (2015) 225–233
10. Manthey, N., Peñaloza, R., Rudolph, S.: Efficient axiom pinpointing in EL using SAT technology. In Lenzerini, M., Peñaloza, R., eds.: Proc. 29th Int. Workshop on Description Logics (DL'16). Volume 1577 of CEUR Workshop Proceedings., CEUR-WS.org (2016)
11. Arif, M.F., Mencía, C., Marques-Silva, J.: Efficient MUS enumeration of Horn formulae with applications to axiom pinpointing. CoRR **abs/1505.04365** (2015)
12. Arif, M.F., Mencía, C., Ignatiev, A., Manthey, N., Peñaloza, R., Marques-Silva, J.: BEACON: an efficient sat-based tool for debugging $\mathcal{EL}^+$ ontologies. In Creignou, N., Berre, D.L., eds.: Proc. 19th Int. Conf. on Theory and Applications of Satisfiability Testing (SAT'16). Volume 9710 of Lecture Notes in Computer Science., Springer (2016) 521–530
13. Bartlett, N.: OSGi In Practice. (2009) Available online at `http://njbartlett.name/osgibook.html`.
14. Horridge, M., Parsia, B., Sattler, U.: Explanation of OWL entailments in Protégé 4. In Bizer, C., Joshi, A., eds.: Proceedings of the Poster and Demonstration Session at the 7th International Semantic Web Conference (ISWC2008), Karlsruhe, Germany, October 28, 2008. Volume 401 of CEUR Workshop Proceedings., CEUR-WS.org (2008)
15. Horridge, M., Parsia, B., Sattler, U.: Laconic and precise justifications in OWL. In Sheth, A., Staab, S., Dean, M., Paolucci, M., Maynard, D., Finin, T., Thirunarayan, K., eds.: Proc. 7th Int. Semantic Web Conf. (ISWC'08). Volume 5318 of LNCS., Springer (2008) 323–338
16. Baader, F., Brandt, S., Lutz, C.: Pushing the $\mathcal{EL}$ envelope. In Kaelbling, L., Saffiotti, A., eds.: Proc. 19th Int. Joint Conf. on Artificial Intelligence (IJCAI'05), Professional Book Center (2005) 364–369
17. Baader, F., Brandt, S., Lutz, C.: Pushing the $\mathcal{EL}$ envelope further. In Clark, K.G., Patel-Schneider, P.F., eds.: Proc. OWLED 2008 DC Workshop on OWL: Experiences and Directions. Volume 496 of CEUR Workshop Proceedings., CEUR-WS.org (2008)
18. Kazakov, Y., Krötzsch, M., Simančík, F.: The incredible ELK: From polynomial procedures to efficient reasoning with $\mathcal{EL}$ ontologies. J. of Automated Reasoning **53**(1) (2014) 1–61