Why Not? Explaining Missing Entailments with Evee

Christian Alrabbaa[1], Stefan Borgwardt[1], Tom Friese[1], Patrick Koopmann[2] and Mikhail Kotlov[1]

[1]*Institute of Theoretical Computer Science, Technische Universität Dresden, 01062 Dresden, Germany*

[2]*Department of Computer Science, Vrije Universiteit Amsterdam, 1081 Amsterdam, Netherlands*

Abstract

Understanding logical entailments derived by a description logic reasoner is not always straight-forward for ontology users. For this reason, various methods for explaining entailments using justifications and proofs have been developed and implemented as plug-ins for the ontology editor Protégé. However, when the user expects a missing consequence to hold, it is equally important to explain why it does not follow from the ontology. In this paper, we describe a new version of Evee, a Protégé plugin that now also provides explanations for missing consequences, via existing and new techniques based on abduction and counterexamples.

Keywords

Protégé plugin, Explanations, Abduction, Counterexamples

1. Introduction

We present a Protégé plugin for explaining missing entailments from OWL ontologies. The importance of explaining description logic reasoning to end-users has long been understood, and has been studied in many forms over the past decades. Indeed, explainability is one of the main advantages of logic-based knowledge representations over sub-symbolic methods. The first approaches to explain *why* a consequence follows from a Description Logic (DL) ontology were based on step-by-step *proofs* [1, 2], but soon research focused on *justifications* [3, 4, 5] that are easier to compute, but still very useful for pointing out the axioms responsible for an entailment. Consequently, the ontology editor Protégé supports black-box methods for computing justifications for arbitrary OWL DL ontologies [6]. More recently, a series of papers investigated different methods of computing good proofs for entailments in DLs ranging from $\mathcal{EL}$ to $\mathcal{ALCOI}$ [7, 8, 9, 10], and the Protégé plug-ins `proof-explanation` [7] and Evee [11], as well as the web-based application Evonne [12], were developed to make these algorithms available to ontology engineers.

While reasoning can sometimes reveal unexpected entailments that need explaining, very often the problem is not what is entailed, but what is *not* entailed. In order to explain such missing entailments, and offer suggestions on how to repair them, both counterexamples and

abduction have been suggested in the literature. A *counterexample* is a model of the ontology that does not satisfy the entailment, which may be further augmented to focus the attention of the user to the part of the model that is most relevant for explaining the non-entailment [13]. In *abduction*, the non-entailment is explained by means of *hypotheses*, which are sets of axioms that can be added to the ontology in order to entail the missing consequence [14, 15, 16]. However, despite there being a lot of research on these explanation services of both theoretical and more practical form, so far, this tool support has not been integrated into standard ontology tools.

In this paper, we present version 0.2 of Evee, a collection of plugins for the OWL ontology editor Protégé, which now also offers explanations for missing entailments. Those plugins integrate the functionality provided by the external tools Capi and Lethe-Abduction for abduction, as well as the counterexample generation methods discussed in [13]. The explanations are provided by Evee through a new *Missing Entailment Explanation* tab that contains a unified interface for explanations based on both counterexamples and abduction. After specifying the missing entailment(s) and optionally a vocabulary for the explanation, the user can choose between different non-entailment explanation algorithms, which then provide either a graphical representation of a counterexample, or a list of different hypotheses to fix the missing entailments. Evee 0.2 has been tested with Java 8, OWL API 4.5.20, and Protégé 5.5.0, and can be downloaded and installed following the instructions at https://github.com/de-tu-dresden-inf-lat/evee. The new plugins depend on the external libraries Capi,[1] Spass,[2] and Lethe-Abduction.[3] This paper also presents a new method for counterexample generation in Section 3.1 and new post-processing steps to make the output of Capi and Lethe-Abduction easier to interpret (see Section 4).

We describe the new Missing Entailment tab of Evee in the next section, before explaining in detail the different explanation services and how they are accessed through the user interface. In Section 5, we explain how developers can use the infrastructure of the Evee Java libraries to provide new explanation services for missing entailments. Additional proofs can be found in the technical report [17].

2. Explanations for Non-Entailments

We assume the reader to be familiar with the syntax and semantics of DLs [18]. The use case of our plugins is the following: we have an active ontology $\mathcal{O}$ opened in the ontology editor Protégé, and there is a set of axioms $\mathcal{P}$ that does not follow from $\mathcal{O}$, i.e. $\mathcal{O} \not\models \mathcal{P}$. The user may also specify a vocabulary Σ to be used for the explanations, which is in particular useful for the abduction services. The *evee-protege-core* component provides the core functionality to specify $\mathcal{P}$ and Σ and extension points for the actual explanation plug-ins. After installing the core plugin, a new tab is available via *Window → Tabs → Missing Entailment Explanation.*

Figure 1 shows this tab in action. It is divided into three major parts: In the upper part, one of the installed *missing entailment explanation services* can be chosen, the computation process can be started, and general information is displayed. On the left, the missing entailment and

---

[1] https://lat.inf.tu-dresden.de/~koopmann/CAPI
[2] https://www.mpi-inf.mpg.de/departments/automation-of-logic/software/spass-workbench/
classic-spass-theorem-prover
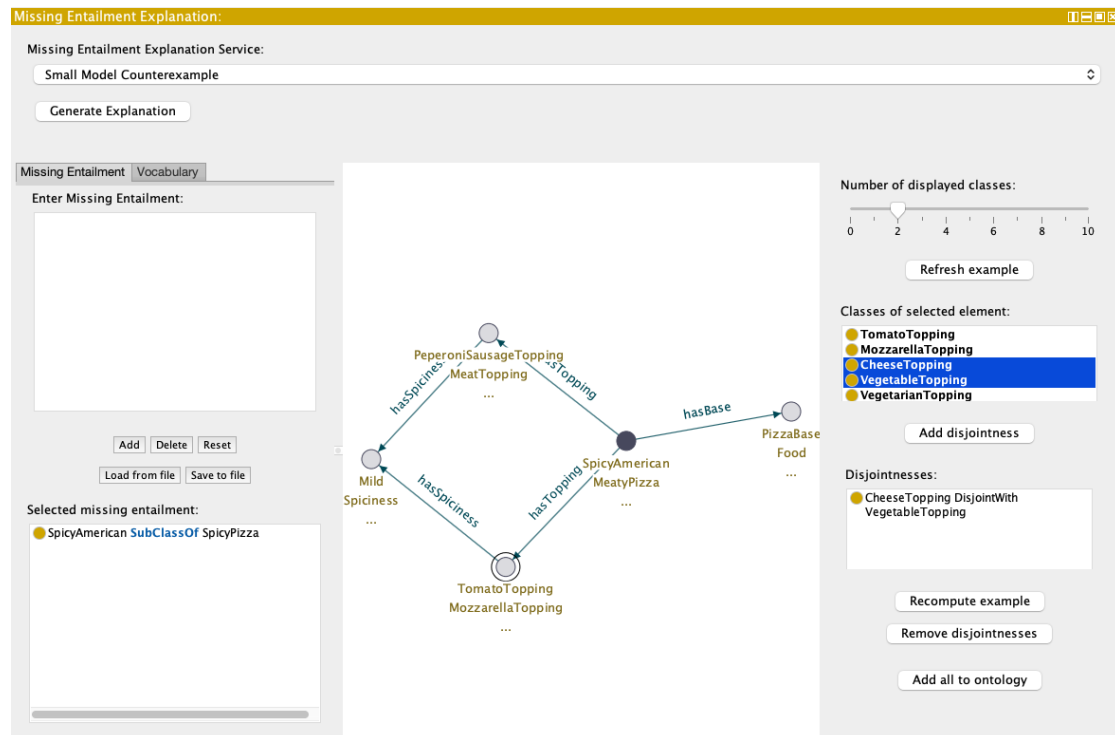[3] https://lat.inf.tu-dresden.de/~koopmann/LETHE-Abduction

Figure 1: A counterexample generated for *SpicyAmerican* ⊑ *SpicyPizza* in an incomplete version of the pizza ontology. The example illustrates two problems with the ontology: The circled element at the bottom shows that *MozarellaTopping* and *TomatoTopping*, as well as *CheeseTopping* and *VegetableTopping* are not disjoint from one another.

vocabulary can be entered, and in the center, the explanation will be displayed. The explanation view depends on the selected explanation service (see Sections 3 and 4). If the entered missing entailment and vocabulary are not supported by the service, the *Generate Explanation* button at the top will be disabled and an explanatory message will be shown.

How missing entailments are entered can be see on the left in Figure 1. The text field at the top can be used to enter individual axioms. The buttons in the middle allow the user to *add* an axiom to the list below, *remove* a selected axiom, or *reset* the whole list. Only *OWL logical axioms* are allowed, e.g. subclass-, equivalence-, and disjointness axioms and assertions. The selected missing entailment can also be saved to or loaded from an OWL ontology file, which can be useful for demonstration purposes.

By selecting the *Vocabulary* tab, the users can restrict the vocabulary used by the explanations, which can be seen in Figure 2. Here, the *Ontology Vocabulary* of the currently active ontology can be accessed via the class hierarchy, object property hierarchy, and list of individuals. The vocabulary of the explanation will be restricted to the names in the tab *Permitted Vocabulary* on the bottom, while *Forbidden Vocabulary* shows the remaining names of the ontology vocabulary. Depending on the currently opened tabs, the arrows and the button *Add missing entailment vocabulary* can be used to add or remove names to and from the selected vocabulary. Again,

the permitted vocabulary can be saved to and loaded from an external file. By default, the whole ontology vocabulary is *permitted*, but this can be changed in the plug-in preferences at *Preferences → Explanations → Missing Entailment → General.*

While the explanation is generated, a progress window is used to indicate the computation status and show additional information. The computation can be canceled by closing the window or clicking the *Cancel* button. This will show a separate cancelation window while the computation is being terminated.

As a running example to illustrate the different explanation services, we consider a modified, incomplete version of the Pizza ontology.[4] This version is missing some axioms to make it entail *SpicyAmerican ⊑ SpicyPizza*. It will turn out that other things are missing in this ontology as well, and the plugin will help the user in adding those missing parts.

3. Counterexamples

The first obvious way to explain the missing entailment is to show an example of a SpicyAmerican that is not a SpicyPizza, as shown in Figure 1. Evee 2.0 includes two plug-ins that provide counterexample generation services: the *Small Model Counterexample Generator* and the *Relevant Counterexample Generator* using ELK [19]. For a missing entailment $C \sqsubseteq D$, a counterexample is a model of the ontology that contains an element that belongs to C, but does not belong to D. The plug-ins visualize counterexamples as directed graphs, where the nodes are individuals labeled by concept names and the edges are labeled by role names. The Small Model Counterexample Generator is developed for $\mathcal{EL}_\perp$, which supports disjointness axioms. This generator generates complete models, but tries to reduce the number of elements to keep the model small. The ELK Relevant Counterexample Generator instead focuses on relevant fragments of models, using the methods described in [13]. It was developed for the description logic $\mathcal{EL}$. Both generators require a single GCI to be entered as non-entailment, and support arbitrary vocabularies Σ. Since we only explain non-entailed GCIs, the counterexample generators ignore the ABox of the active ontology, and only consider the TBox. The generated counterexample only shows names from Σ. We first describe the counterexample view, before explaining the different methods in detail.

To visualize the counterexamples, we use GraphStream,[5] a Java library for modeling, analyzing and visualizing graphs. Its functionality allows not only to visualize models, but also to dynamically make changes to models when the ontologies change. In the generated graphs, domain elements are depicted as circles. We highlight elements that are of particular importance for understanding the generated model. For instance, each counterexample contains a *root element*, marked in black, which satisfies the concept on the left-hand side of the GCI to be explained. For readability, only some of the concept names for each domain element are shown, whose number can be adapted using the *Number of displayed classes* slider on the right panel of the counterinterpretation view. When the user selects a node, the *Classes of selected element* list in the right panel displays all the concept names to which the selected element belongs. In the

---

node selected in Figure 1, the user notices in this way an element that is both a *TomatoTopping* and a *MozzarellaTopping*, pointing at another bug in the ontology—but more on this later.

The graphical model view allows zoom to facilitate exploring large graphs, and users can move the nodes of the graph. Dragging the mouse over the background canvas navigates through the graph. To make the graphical representation of a counterexample more informative, we display concept names in the order from more specific to less specific. In Figure 1, we display that the root element belongs only to *SpicyAmerican* and *MeatyPizza*, but it implicitly is also a *Pizza*, a *Food*, and ultimately a *DomainThing*, since the first two classes are subsumed by them. However, if we instead displayed in the graph that this element is a *DomainThing*, we would give no useful information about the element.

As in the present example, the visualized model can also reveal missing disjointness axioms in the ontology. As already noticed, in Figure 1, the selected element is both a *MozzarellaTopping* and a *TomatoTopping*. The reason is a missing disjointness between *CheeseTopping* and *VegetableTopping*. The right panel allows the user to add new disjointness axioms as needed and visualize the result. For this, the user selects the corresponding concept names in the *Classes of selected element* list and presses *Add disjointnesses*. A disjointness axiom with the selected names is then added to the *Disjointnesses* list, as shown in the figure. By pressing the *Recompute example* button, the user gets shown an updated model with the new disjointness applied. If the user is not satisfied with the changes to the model resulting from the new axioms, they can be deleted using the *Remove disjointnesses* button. Finally, axioms from the *Disjointnesses* list can be added to the active ontology with a click of the *Add all to ontology* button.

3.1. Small Model Counterexample Generator

In this explanation service, counterexamples are generated using a tableau-based algorithm for the description logic $\mathcal{EL}_\perp$. Given a GCI $C \sqsubseteq D$, the algorithm first initializes an ABox $\mathcal{A}$ containing as only axiom $C(a^*)$, where a^* is a fresh individual name. Next, it adds $D \sqsubseteq B^*$ to the TBox, where B^* is a fresh concept name, and normalizes [18] the TBox. Note that $\mathcal{T} \models C \sqsubseteq D$ iff $\mathcal{T} \cup \{D \sqsubseteq B^*\} \models C \sqsubseteq B^*$. Thus, if $a^{*\mathcal{I}} \notin B^{*\mathcal{I}}$ in the generated model $\mathcal{I}$, then $a^{*\mathcal{I}} \notin D^{\mathcal{I}}$. Therefore, the generated model is a counterexample iff $a^{*\mathcal{I}} \notin B^{*\mathcal{I}}$ [18].

The model of the ontology is obtained using a complete and clash-free ABox $\mathcal{A}'$ obtained from the ABox $\mathcal{A}$ by an exhaustive application of the expansion rules from Table 1. The $\sqsubseteq$-rule is almost identical to the similar rule in algorithms for $\mathcal{ALC}$. The only difference is that it takes into account the structure of the normalized TBox. It becomes applicable only to concept assertions with concept names or with a concept name under an existential restriction. The $\sqcap$- and $\exists_1$-rules are designed to add assertions that make the $\sqsubseteq$-rule applicable. For individual names having a successor belonging to some concept name, the $\exists_1$-rule creates a concept assertion with this concept name under an existential restriction. The $\sqcap$-rule breaks conjunctions into simpler assertions.

To keep the model small, we reuse existing individuals as successors when trying to satisfy existential role restrictions. Before reusing an individual name, a consistency check is performed, so that the rule cannot introduce any inconsistency. Moreover, we only reuse an individual as successor if this does not make the root element an instance of B^*, since the aim is to construct a counterexample for $C \sqsubseteq B^*$.

Table 1

Expansion rules of the tableau method generating small counterexamples for $\mathcal{EL}_\perp$. Here, $\text{At}(D)$ refers to the conjuncts of the concept D, or to the singleton set $\{D\}$ if D is not a conjunction.

$\sqcap$-**rule**	**if** $\mathcal{A}$ contains $D(a)$, but not $C(a)$, $C \in \text{At}(D)$ **then** $\mathcal{A} \longrightarrow \mathcal{A} \cup \{C(a)\}$
$\exists_1$-**rule**	**if** $\mathcal{A}$ contains $r(a,b)$ and $A(b)$, A is a concept name or $\top$, but not $\exists r.A(a)$
	$\mathcal{A} \longrightarrow \mathcal{A} \cup \{\exists r.A(a)\}$
$\exists_2$-**rule**	**if** $\mathcal{A}$ contains $\exists r.E(a)$, but there is no b s.t. $r(a,b)$ and $E(b)$
	if there is some c, s.t. $\mathcal{T} \cap \mathcal{A} \cup \{r(a,c), E(c)\}$ is consistent and does not entail $B^*(a^*)$ **then** $\mathcal{A} \longrightarrow \mathcal{A} \cup \{r(a,c), E(c)\}$
	else $\mathcal{A} \longrightarrow \mathcal{A} \cup \{r(a,d), E(d), \top(d)\}$, where d is new in $\mathcal{A}$
The $\sqsubseteq$-rule	**if** $A(a) \in \mathcal{A}$, $A \sqsubseteq B \in \mathcal{T}$ or $\{A_1(a), A_2(a)\} \subseteq \mathcal{A}$, $A_1 \sqcap A_2 \sqsubseteq B \in \mathcal{T}$ or $\exists r.A(a) \in \mathcal{A}$, $\exists r.A \sqsubseteq B \in \mathcal{T}$ and $B(a) \notin \mathcal{A}$.
	then $\mathcal{A} \longrightarrow \mathcal{A} \cup \{B(a)\}$

The expansion rules are applied exhaustively, but the $\exists_2$-rule is applied only if no other rule is applicable. This restriction reduces the number of applications of the $\exists_2$-rule, and consequently the number of individuals added. The algorithm iteratively applies the rules until no more rule is applicable, and then translates the resulting ABox into an interpretation in the usual way. The correctness of the algorithm is shown in the technical report [17].

3.2. Relevant Counterexample Generator

A more focussed explanation to missing entailments is provided by the *Relevant Counterexample Generator*. *Relevant counterexamples* explain missing entailments by showing relevant parts of the models of $\mathcal{EL}$ ontologies, where this time *canonical models* [18] are used. Canonical models have two properties that are beneficial for explanations. First, they reuse domain elements, i.e. when a concept C appears multiple times in a TBox $\mathcal{T}$, the substructure of the canonical model $\mathcal{I}_\mathcal{T}$ satisfying C is reused. This makes $\mathcal{I}_\mathcal{T}$ a compact interpretation. Second, for any non-entailment η, $\mathcal{T} \not\models \eta$ iff $\mathcal{I}_\mathcal{T} \not\models \eta$, and hence $\mathcal{I}_\mathcal{T}$ directly serves as a counterexample. However, the size of these models can still be large. To overcome this, we focus on certain parts of the model, since in general not the entire model is relevant for the explanation of the current η.

We distinguish four types of relevance as shown in [13], which define the α-, β-, Δ-, and $\bar{\Delta}$-*relevant parts* of $\mathcal{I}_\mathcal{T}$. One possible explanation for $\mathcal{T} \not\models C \sqsubseteq D$ is to show the user an element that satisfies C, does not satisfy D, and satisfies all axioms in $\mathcal{T}$. This element serves as a *witness* for the non-entailment, and together with its required successors forms the α-*relevant part* of the canonical model. Another possibility is to contrast C with D, by including also a representative element satisfying D, which gives rise to the β-*relevant part* of the canonical model.

The Δ-relevant part is a refinement of the β-relevant part that focuses on the conditions that are imposed by the ontology on D, but not on C. This allows for an "explanation by contradiction" as follows. If $\mathcal{T} \models C \sqsubseteq D$, then every subsumer E of D must also subsume C. However, there is a model of $\mathcal{T}$ (the canonical model) in which there is an element of C that intuitively does not satisfy some condition E that is satisfied by every element of D. Hence, C cannot be subsumed by D w.r.t. $\mathcal{T}$. Therefore, the Δ-relevant part contains only those
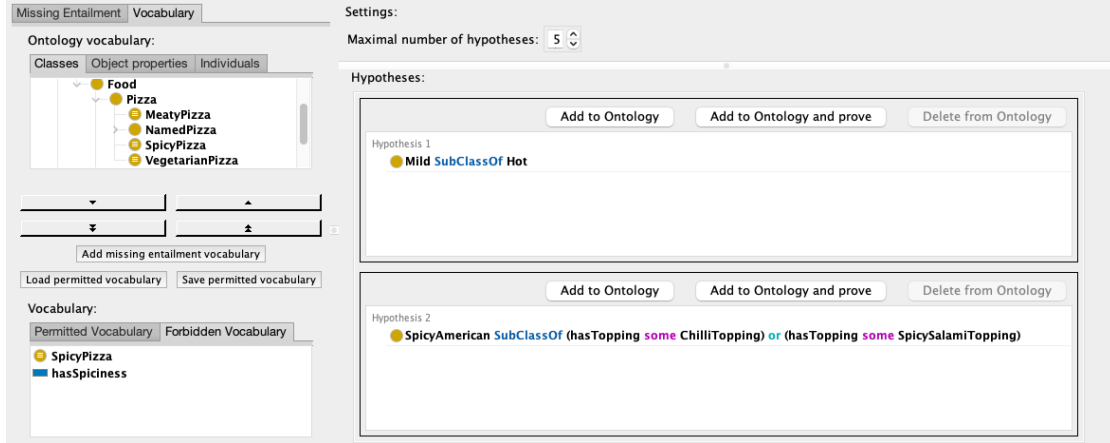
Figure 2: Abduction results from Lᴇᴛʜᴇ for *SpicyAmerican* ⊑ *SpicyPizza* with forbidden symbols *SpicyPizza* and *hasSpiciness*.

elements illustrating the contrasting conditions E, e.g. r-successors (not) satisfying F in case that $E = \exists r.F$. The $\bar{\Delta}$-relevant part is a further refinement of the Δ-relevant part that tries to generalize these conditions E. For example, if $\mathcal{T} \models D \sqsubseteq E$, $\mathcal{T} \not\models C \sqsubseteq E$ and $E = \exists r.\exists r.\exists r.F$, then it is sufficient to consider $\exists r.\top$ instead of E, assuming that $\mathcal{T} \not\models C \sqsubseteq \exists r.\top$. For more details we refer the reader to [13].

4. Abduction

Counterexamples always focus only on one model, and they do not necessarily make it obvious what needs to be done to fix a missing entailment. This is where the explanation services based on abduction come into play. For our running example, we show an explanation based on abduction in Figure 2. Evee 0.2 includes two plug-ins based on *abduction*, namely the *Complete Signature-Based Abduction solver* based on Lᴇᴛʜᴇ [20][6] and the *Connection-Minimal Abduction solver* utilizing CAPI [16].[7] Given a non-entailment $\mathcal{O} \not\models \mathcal{P}$, abduction computes a set of *hypotheses* $\mathcal{H}$, which are sets of axioms such that $\mathcal{O} \cup \mathcal{H} \models \mathcal{P}$. Without further restrictions, $\mathcal{P}$ is already a hypothesis, which is why usually additional constraints on the solution space are given. *Signature-based abduction* [15] relies on a user-given vocabulary. The signature-based abduction service computes a set of alternative hypotheses using only names from the vocabulary, such that any other such hypothesis can be obtained by strengthening or combining those hypotheses. In contrast, CAPI computes hypotheses satisfying a minimality criterion called *connection-minimality* [16], with the aim of focussing on those hypotheses that have a more direct connection to the observation. The signature-based explanations support the DL $\mathcal{ALC}$, observations can be a mix of several ABox and TBox axioms, and the hypotheses can make arbitrary use of DL constructs, which in particular means that the result can be an unbounded sequence of hypotheses. Connection minimal explanations support $\mathcal{EL}$ ontologies,
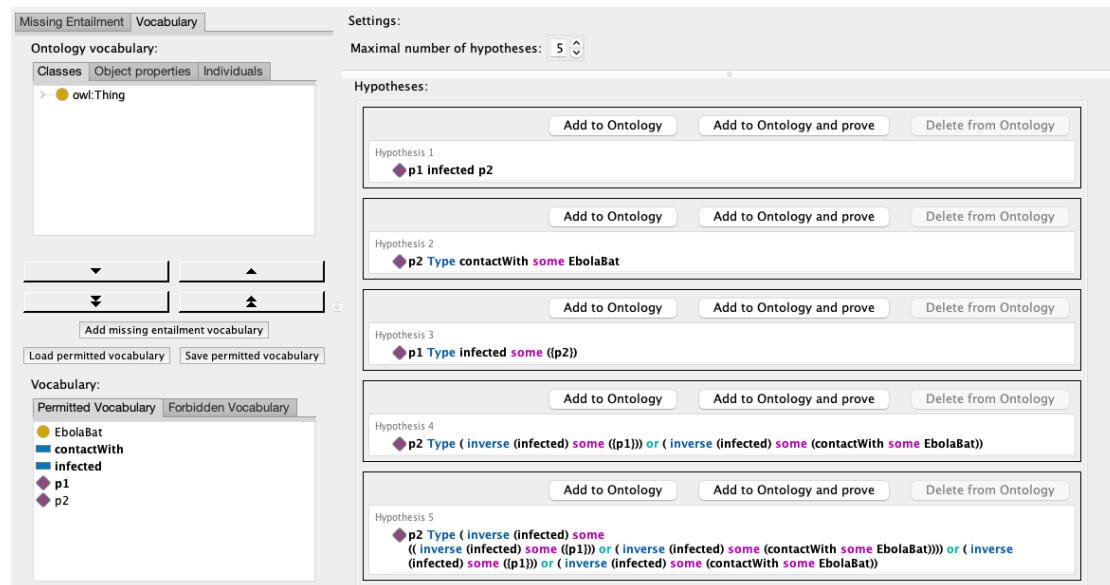
---

[6]https://lat.inf.tu-dresden.de/~koopmann/LETHE-Abduction
[7]https://lat.inf.tu-dresden.de/~koopmann/CAPI

Figure 3: An abduction result giving possible explanations for why patient *p2* is an *EbolaPatient*. This is based on the example used in [14].

entailments consisting of a single GCI, and hypotheses are always without role restrictions. To use CAPI, the FOL theorem prover SPASS needs to be installed separately. We require an adapted version of SPASS, which can be installed following the instructions on the web page of CAPI.[8]

After computing an explanation with an abduction solver, one or more hypotheses will be displayed in a list, as shown in Figures 2 and 3. Depending on the input and the algorithm, the number of results may differ and may even be infinite. Therefore, the user can specify the number of new results that are added to the list whenever the *Generate Explanation* button is clicked again. Using additional buttons shown at each hypothesis, the user can then easily add the hypothesis to the ontology (to repair the non-entailment) and get an explanation why the hypothesis entails the non-entailment, using the proof functionality provided by the `proof-explanation` and Evee 0.1 plug-ins [7, 11]. The third button can be used to revert the changes to the ontology. Each service resets the displayed results if any changes are made to the active ontology, unless these changes are made via these *Add* and *Delete* buttons.

4.1. Complete Signature-Based Abduction

Signature based hypotheses are computed by the abduction extension of the external library LETHE [14, 20]. We extended the original method by an additional, equivalence-preserving simplification step to make the hypotheses more user-friendly. The method computes so-called *complete signature-based hypotheses*, which are hypotheses that are fully in the signature, and which generalize any other possible such hypothesis. This is only possible by using disjunctions

---

[8]When using the CAPI abduction plug-in for the first time, it will ask for the directory that SPASS was installed to. This directory can later be changed in the Protégé preferences, see Section 4.2.

and least fixpoint operators, which is why the output of this method is a disjunction of the form $\bigvee_{i=1}^{n} \left(\bigwedge_{j=1}^{m} \alpha_{i,j} \right)$, with each $\alpha_{i,j}$ an $\mathcal{ALCOI}\mu$ axiom. Intuitively, each disjunct is an alternative hypothesis, but their axioms may include *least fixpoint concepts* of the form $\mu X.C[X]$ [21]. To obtain from this disjunction a sequence of hypotheses that can be displayed in Protégé, the fixpoint concepts need to be *unraveled*. This is done in order of increasing role depth, i.e. the shallowest hypotheses are shown first. For example, in Figure 3, hypotheses 4 and 5 are obtained by unravelling of the following assertion, followed by some syntactic reformulations:

$$p_2 : \mu X.\exists\mathsf{infected}^-.\big(\exists\mathsf{contactWith}.\mathsf{EbolaBat} \sqcup \{p_1\} \sqcup X\big)$$

4.2. Capi Abduction solver

The CAPI abduction solver internally relies on the FOL theorem prover SPASS to compute the solutions to an abduction problem. In particular, based on a translation into first-order logic clauses, SPASS computes a set of prime implicates, which are then used by the Java component of the tool to construct the different hypotheses (see [16] for details). The Protégé plugin takes some additional input parameters that can be configured by the user. By default, SPASS stops generating prime implicates after a time limit of 10 seconds is reached. This is usually sufficient to obtain a large set of hypotheses, but if the results are unsatisfactory, the time limit can be changed under *Preferences → Explanations → Missing Entailment → Connection-Minimal Abduction (CAPI)*. Further options concern the post-processing of solutions generated by SPASS, which were not included in the original implementation presented in [16], but later added for convenience: 1) explanations can be simplified by removing redundant axioms, 2) axioms can be simplified by removing redundant conjuncts or disjuncts, and 3) hypotheses can be ordered by specificity, which means: if one hypothesis implies another one, the implied hypothesis is shown later. Without these post-processing steps, hypotheses may be long and generally contain long lists of conjunctions, which is why the optimizations are turned on by default. On the other hand, by deactivating all post-processing steps, we obtain hypotheses that are faithful to the method described in [16].

5. Adding New Non-Entailment Explanation Services

For developers who want to add their own non-entailment explanation services, the module *evee-protege-core* provides two new extension points for Protégé plug-ins:

        `de.tu_dresden.inf.lat.evee.nonEntailment_explanation_service`
for explanation services and

            `de.tu_dresden.inf.lat.evee.nonEntailment_preferences`
for managing plug-in-specific preference settings.

The preferences extension point is simple: One implements the interface `PreferencesPanel` provided by Protégé, and the resulting panel will be displayed in a tabbed pane accessible via *Preferences → Explanations → Missing Entailment*. Using the explanation service extension point requires a few more steps. Essentially, an explanation service needs to provide the non-entailment explanations as a Java Stream and visualize the elements of this stream in Protégé.

To facilitate this for abduction and counterexamples, we provide two abstract base classes for abduction and counterexample services.

The main interface for explanation services is `IOWLNonEntailmentExplainer`, whose most important methods are `supportsExplanation()` and `generateExplanations()`. The first method determines whether the *Generate Explanation* button should be enabled or disabled. Since this method is called whenever the input changes, its implementation should not be computationally expensive. The second method returns the explanation in the form of a `Stream<Set<OWLAxiom>>`, where each set represents a single explanation for the missing entailment.[9] We use streams to accommodate a potentially infinite number of explanations, as in the case of signature-based abduction.

On top of these generic methods, `INonEntailmentExplanationService` provides functionality to connect to the user interface. The method `computeExplanation()` is called when the *Generate Explanation* button is clicked, and `cancel()` is called when the user wants to cancel. The explanation service can also use an `IProgressTracker` to send information to the loading window and an `IExplanationGenerationListener` to send events to the main tab. To the loading window, one can send the current progress as well as a String describing the current computation status. The events for the listener can have an `ExplanationEventType` of COMPUTATION_COMPLETE, RESULT_RESET, WARNING, or ERROR. This allows the explanation service to display new results, clear the shown result, or display warnings or errors, respectively. The main tab ultimately requires the result in the form of a `java.awt.Component`, which is retrieved via the method `getResult` right after an event of type COMPUTATION_COMPLETE is received. This way, each service enjoys a great degree of freedom in displaying its explanation. We already provide pre-built functionality for abduction and counterexample services, as described in the following sections.

5.1. Abstract Counterexample Generation Service

The class `AbstractCounterexampleGenerationService` contains all functionality related to the visualization of counterexamples and implements all methods of the `INonEntailment-ExplanationService` interface. Classes extending `AbstractCounterexampleGenera-tionService` differ primarily in the used counterexample generator, which must be specified in the constructor using the method `setCounterexampleGenerator()`.

Each counterexample generator implements the `IOWLCounterexampleGenerator` interface. The interface extends `IOWLNonEntailmentExplainer` by `generateModel()` and `getMarkedIndividuals()`. The model returned by `generateModel()` is represented using a set of `OWLIndividualAxioms`. Each of those should be an instance of either `OWLClass-AssertionAxiom` or `OWLObjectPropertyAssertionAxiom`, which specify the content of the classes and properties in the interpretation. These axioms are also returned by the method `generateExplanations()` of the interface `IOWLCounterexampleGenerator`. Finally, using the method `getMarkedIndividuals()`, the service can specify individual names that will be highlighted in the visualization of the model.

---

[9]For abduction, these sets are the hypotheses, and for counterexamples they are sets of assertions that describe models.

As an example of how the abstract counterexample generator operates, consider again the algorithm described in Section 3.1. This algorithm is implemented in a separate counterexample generator and executed when `generateModel()` is called. Afterwards, the abstract counterexample generator sends an `ExplanationEvent` of type `COMPUTATION_COMPLETE` to the main tab. The resulting counter example is then provided to the main tab via the method `generateExplanations()`.

5.2. Abstract Abduction Solver

The class `AbstractAbductionSolver` is used by both of the plug-ins presented in Section 4. The main responsibilities of this class are caching the results computed for a specific input, creating and maintaining the actual result component that is displayed to the user, and handling user input when any of the *Add-* or *Delete*-buttons of a hypothesis are clicked (see Figure 3). The class is generic in order to facilitate the caching of different kinds of results for each implementing solver via its generic type parameter. Caching is not done automatically by the abstract solver class. Instead, the implementing solver can use the methods `checkResultInCache`, `saveResultToCache` and `loadResultFromCache`.

In contrast to these user-experience-related functionalities, the actual computation of the missing entailment explanation is left to the individual implementations of the abstract class. As explained above, implementing the interface `IOWLNonEntailmentExplainer` requires providing a stream of explanations via the method `generateExplanations()`, i.e. a stream of sets of OWLAxioms, where each set represents a single hypothesis. This method will ultimately be called by the `AbstractAbductionSolver` when creating the list of non-entailment explanations that is shown to the user.

6. Conclusion

We believe that our plug-ins are an important step towards making reasoning more understandable to ontology users. The implementation is still relatively new and there are little performance issues that need to be solved. We hope that our framework will encourage other developers to implement their own explanation services in Evee. In addition to further improving Evee, we would like to evaluate our plug-ins in a user study. It would also be interesting to investigate whether Evee can be used to improve university-level teaching on ontologies and description logics.

Acknowledgments

References

[1] A. Borgida, E. Franconi, I. Horrocks, Explaining ALC subsumption, in: W. Horn (Ed.), ECAI 2000, Proceedings of the 14th European Conference on Artificial Intelligence, IOS Press, 2000, pp. 209–213. URL: http://www.frontiersinai.com/ecai/ecai2000/pdf/p0209.pdf.

[2] D. L. McGuinness, Explaining Reasoning in Description Logics, Ph.D. thesis, Rutgers University, NJ, USA, 1996. doi:10.7282/t3-q0c6-5305.

[3] F. Baader, R. Peñaloza, B. Suntisrivaraporn, Pinpointing in the description logic EL^+, in: J. Hertzberg, M. Beetz, R. Englert (Eds.), KI 2007: Advances in Artificial Intelligence, 30th Annual German Conference on AI, volume 4667 of *Lecture Notes in Computer Science*, Springer, 2007, pp. 52–67. doi:10.1007/978-3-540-74565-5_7.

[4] M. Horridge, Justification Based Explanation in Ontologies, Ph.D. thesis, University of Manchester, UK, 2011. URL: https://www.research.manchester.ac.uk/portal/files/54511395/FULL_TEXT.PDF.

[5] S. Schlobach, R. Cornet, Non-standard reasoning services for the debugging of description logic terminologies, in: G. Gottlob, T. Walsh (Eds.), IJCAI-03, Proceedings of the Eighteenth International Joint Conference on Artificial Intelligence, Morgan Kaufmann, 2003, pp. 355–362. URL: http://ijcai.org/Proceedings/03/Papers/053.pdf.

[6] M. Horridge, B. Parsia, U. Sattler, Explaining inconsistencies in OWL ontologies, in: L. Godo, A. Pugliese (Eds.), Scalable Uncertainty Management, Third International Conference, SUM, Proceedings, volume 5785 of *Lecture Notes in Computer Science*, Springer, 2009, pp. 124–137. doi:10.1007/978-3-642-04388-8_11.

[7] Y. Kazakov, P. Klinov, A. Stupnikov, Towards reusable explanation services in protege, in: A. Artale, B. Glimm, R. Kontchakov (Eds.), Proceedings of the 30th International Workshop on Description Logics, volume 1879 of *CEUR Workshop Proceedings*, CEUR-WS.org, 2017. URL: http://ceur-ws.org/Vol-1879/paper31.pdf.

[8] C. Alrabbaa, F. Baader, S. Borgwardt, P. Koopmann, A. Kovtunova, Finding small proofs for description logic entailments: Theory and practice, in: E. Albert, L. Kovács (Eds.), LPAR 2020: 23rd International Conference on Logic for Programming, Artificial Intelligence and Reasoning, volume 73 of *EPiC Series in Computing*, EasyChair, 2020, pp. 32–67. doi:10.29007/nhpp.

[9] C. Alrabbaa, F. Baader, S. Borgwardt, P. Koopmann, A. Kovtunova, On the complexity of finding good proofs for description logic entailments, in: S. Borgwardt, T. Meyer (Eds.), Proceedings of the 33rd International Workshop on Description Logics (DL 2020), volume 2663 of *CEUR Workshop Proceedings*, CEUR-WS.org, 2020. URL: http://ceur-ws.org/Vol-2663/paper-1.pdf.

[10] C. Alrabbaa, F. Baader, S. Borgwardt, P. Koopmann, A. Kovtunova, Finding good proofs for description logic entailments using recursive quality measures, in: A. Platzer, G. Sutcliffe (Eds.), Automated Deduction - CADE 28 - 28th International Conference on Automated Deduction, volume 12699 of *Lecture Notes in Computer Science*, Springer, 2021, pp. 291–308. doi:10.1007/978-3-030-79876-5_17.

[11] C. Alrabbaa, S. Borgwardt, T. Friese, P. Koopmann, J. Méndez, A. Popovic, On the eve of true explainability for OWL ontologies: Description logic proofs with Evee and Evonne, in: O. Arieli, M. Homola, J. C. Jung, M. Mugnier (Eds.), Proceedings of the 35th International

Workshop on Description Logics (DL), volume 3263 of *CEUR Workshop Proceedings*, CEUR-WS.org, 2022. URL: https://ceur-ws.org/Vol-3263/paper-2.pdf.

[12] J. Méndez, C. Alrabbaa, P. Koopmann, R. Langner, F. Baader, R. Dachselt, Evonne: A visual tool for explaining reasoning with OWL ontologies and supporting interactive debugging, Computer Graphics Forum (2023). doi:https://doi.org/10.1111/cgf.14730.

[13] C. Alrabbaa, W. Hieke, Explaining non-entailment by model transformation for the description logic $\mathcal{EL}$, in: A. Artale, D. Calvanese, H. Wang, X. Zhang (Eds.), Proceedings of the 11th International Joint Conference on Knowledge Graphs, IJCKG, ACM, 2022, pp. 1–9. doi:10.1145/3579051.3579060.

[14] P. Koopmann, W. Del-Pinto, S. Tourret, R. A. Schmidt, Signature-based abduction for expressive description logics, in: D. Calvanese, E. Erdem, M. Thielscher (Eds.), Proceedings of the 17th International Conference on Principles of Knowledge Representation and Reasoning, KR, 2020, pp. 592–602. doi:10.24963/kr.2020/59.

[15] P. Koopmann, Signature-based abduction with fresh individuals and complex concepts for description logics, in: Z. Zhou (Ed.), Proceedings of the Thirtieth International Joint Conference on Artificial Intelligence, IJCAI, ijcai.org, 2021, pp. 1929–1935. doi:10.24963/ijcai.2021/266.

[16] F. Haifani, P. Koopmann, S. Tourret, C. Weidenbach, Connection-minimal abduction in $\mathcal{EL}$ via translation to FOL, in: J. Blanchette, L. Kovács, D. Pattinson (Eds.), Automated Reasoning, Springer International Publishing, Cham, 2022, pp. 188–207. doi:10.1007/978-3-031-10769-6_12.

[17] C. Alrabbaa, S. Borgwardt, T. Friese, P. Koopman, M. Kotlov, Why not? Explaining missing entailments with Evee (Technical report), 2023. doi:10.48550/arXiv.2308.07294.

[18] F. Baader, I. Horrocks, C. Lutz, U. Sattler, An Introduction to Description Logic, Cambridge University Press, 2017. doi:10.1017/9781139025355.

[19] Y. Kazakov, M. Krötzsch, F. Simancik, The incredible ELK - from polynomial procedures to efficient reasoning with $\mathcal{EL}$ ontologies, J. Autom. Reason. 53 (2014) 1–61. doi:10.1007/s10817-013-9296-3.

[20] P. Koopmann, LETHE: forgetting and uniform interpolation for expressive description logics, Künstliche Intell. 34 (2020) 381–387. doi:10.1007/s13218-020-00655-w.

[21] D. Calvanese, G. D. Giacomo, M. Lenzerini, Reasoning in expressive description logics with fixpoints based on automata on infinite trees, in: T. Dean (Ed.), Proceedings of the Sixteenth International Joint Conference on Artificial Intelligence, IJCAI, Morgan Kaufmann, 1999, pp. 84–89. URL: http://ijcai.org/Proceedings/99-1/Papers/013.pdf.