The Protégé OWL Plugin: An Open Development Environment for Semantic Web Applications

Holger Knublauch, Ray W. Fergerson, Natalya F. Noy and Mark A. Musen

Stanford Medical Informatics, Stanford School of Medicine
251 Campus Drive, Stanford, CA 94305-5479
holger@smi.stanford.edu

Abstract. We introduce the OWL Plugin, a Semantic Web extension of the Protégé ontology development platform. The OWL Plugin can be used to edit ontologies in the Web Ontology Language (OWL), to access description logic reasoners, and to acquire instances for semantic markup. In many of these features, the OWL Plugin has created and facilitated new practices for building Semantic Web contents, often driven by the needs of and feedback from our users. Furthermore, Protégé's flexible open-source platform means that it is easy to integrate custom-tailored components to build real-world applications. This document describes the architecture of the OWL Plugin, walks through its most important features, and discusses some of our design decisions.

1 Introduction

Efficient development tools are a prerequisite for the wide adoption of a new technology. For example, visual Web design tools like DreamWeaver have significantly reduced the development costs of Internet content, and have brought Web technology to the fingertips of people who are not familiar with the details of HTML. The concept of the Semantic Web [3] is often regarded as the next "big" technology leap for the Internet. Now that the Web Ontology Language (OWL) [16] has been officially standardized, it is the task of tool builders to explore and provide suitable infrastructures that help make the Semantic Web vision a reality. This document reports on our work and results in the construction of such a tool.

Our goal was to build an extensible tool with support for the most commonly needed features for the development of Semantic Web applications. *Ontologies* play a central role in the Semantic Web: They provide formal models of domain knowledge that can be exploited by intelligent agents. As a result, a development tool for Semantic Web applications should provide services to access, visualize, edit, and use ontologies. Furthermore, since ontologies may be notoriously hard to build [13], a tool should provide intelligent assistance in ontology construction and evolution. Finally, since the Semantic Web is open to so many potential application areas, a tool should be customizable and extensible.

We have developed the OWL Plugin, an extension of Protégé. Protégé [4, 8] is an open platform for ontology modeling and knowledge acquisition. The OWL Plugin can

be used to edit OWL ontologies, to access description logic (DL) reasoners, and to acquire instances for semantic markup. As an extension of Protégé, the OWL Plugin profits from the benefits of a large user community, a library of reusable components, and a flexible architecture. The OWL Plugin therefore has the potential to become a standard infrastructure for building ontology-based Semantic Web applications.

This document describes the main features of the Protégé OWL Plugin and illustrates how it can be used and customized to build real-world Semantic Web applications. Section 2 lists requirements and design goals that motivated the development of the OWL Plugin. Section 3 provides background on the Protégé platform and architecture. Section 4 describes how Protégé has been extended with support for OWL, and how other components can be built on top of it. Section 5 shows how the OWL Plugin can be used to edit OWL classes and properties. Section 6 describes features that help evolve, test and maintain ontologies, including intelligent reasoning based on description logics. Section 7 shows how Protégé can serve as an editor of Semantic Web contents (e.g., OWL individuals). Section 8 discusses our results and points at ongoing and future work. This paper assumes that the reader is familiar with OWL, but not necessarily with Protégé.

2 Requirements and Design Goals

The nature of the Semantic Web and its languages has implications for tool builders. A rather well-known aspect is that the cryptic and error-prone RDF syntax of OWL [16] makes it extremely hard to build valid OWL ontologies manually. Graphical ontology editing tools with a simple syntax and built-in validation mechanisms could significantly improve this situation.

A more critical aspect is that the Semantic Web is based on ontologies, and ontologies are notoriously difficult to build [13]. One reason for this difficulty is that ontologies are formal models of human domain knowledge. While human knowledge is often tacit and hard to describe in formal models, there is also no single correct mapping of knowledge into discrete structures. Although some rules of thumb exist that facilitate selected ontology-design tasks (e.g., [13, 15]), there are hardly any comprehensive ontology development methods in routine use. As an alternative to systematic and comprehensive guidelines, development tools should at least support rapid turn-around times to encourage ontology evolution and iteration. User interfaces should simplify and accelerate common tasks and at the same time encourage best practices and design patterns. Furthermore, tools should expose somewhat intelligent assistance for ontology builders, and point out obvious modeling errors. Another important criteria is that ontologies can become very large, so that tools need to be scalable, and should support the coordination of efforts among multiple people.

A final issue for Semantic Web development tools is that they should be *open*. First, they should be open-source so that interested people can more easily try and understand them. Second, tools should also be open to project-specific customizations and extensions. Nobody knows where the Semantic Web will be in the future, and it may offer boundless opportunities for innovative services, methods, and applications. An open tool that allows for the integration of these future components more easily may sig-

nificantly reduce development costs. For example, developers of OWL reasoners could integrate their reasoning component into an OWL editor so that they can much easier experiment with different ontologies. Also, developers of Web services could reuse the tool's infrastructure without having to write their own base platform first.

3 Protégé

Protégé is an open-source tool developed at Stanford Medical Informatics. It has a community of thousands of users. Although the development of Protégé has historically been mainly driven by biomedical applications [4], the system is domain-independent and has been successfully used for many other application areas as well.

Like most other modeling tools, the architecture of Protégé is cleanly separated into a "model" part and a "view" part. Protégé's *model* is the internal representation mechanism for ontologies and knowledge bases. Protégé's *view* components provide a user interface to display and manipulate the underlying model.

Protégé's *model* is based on a simple yet flexible metamodel [11], which is comparable to object-oriented and frame-based systems. It basically can represent ontologies consisting of classes, properties (slots), property characteristics (facets and constraints), and instances. Protégé provides an open Java API to query and manipulate models. An important strength of Protégé is that the Protégé metamodel itself is a Protégé ontology, with classes that represent classes, properties, and so on. For example, the default class in the Protégé base system is called `:STANDARD-CLASS`, and has properties such as `:NAME` and `:DIRECT-SUPERCLASSES`. This structure of the metamodel enables easy extension and adaption to other representations [12]. For example, we have extended this metamodel to handle UML and OWL.

Using the *views* of Protégé's user interface, ontology designers basically create classes, assign properties to the classes, and then restrict the properties' facets at certain classes. Using the resulting ontologies, Protégé is able to automatically generate user interfaces that support the creation of individuals (instances). For each class in the ontology, the system creates one *form* with editing components (*widgets*) for each property of the class. For example, for properties that can take single string values, the system would by default provide a text field widget. The generated forms can be further customized with Protégé's form editor, where users can select alternative user interface widgets for their project. In addition to the predefined library of user interface widgets, Protégé has a flexible architecture that enables programmers to develop custom-tailored widgets, which can then be plugged into the core system. Another type of plugin supports full-size user interface panels (*tabs*) that can contain arbitrary other components. In addition to the collection of standard tabs for editing classes, properties, forms and instances, a library of other tabs exists that perform queries, access data repositories, visualize ontologies graphically, and manage ontology versions.

Protégé currently can be used to load, edit and save ontologies in various formats, including CLIPS, RDF, XML, UML and relational databases. Recently, we have added support for OWL. Our decision to build our system on top of Protégé was driven by various factors. Since ontologies play such an important role in Semantic Web applications, it was straight-forward to take an existing ontology development environment as

a starting point. Extensions to Protégé can benefit from the generic services provided by the core platform, such as an event mechanism, undo capabilities, and a plugin mechanism. By basing the OWL Plugin on top of Protégé, we could also reuse Protégé's client-server-based multi-user mode that allows multiple people to edit the same ontology at the same time. Protégé also provides a highly scalable database back-end, allowing users to create ontologies with hundreds of thousands of classes. Also, there is already a considerable library of plugins which can be either directly used for OWL or adapted to OWL with little effort. Furthermore, the fact that Protégé is open-source also encourages plugin development. Last but not least, Protégé is backed by a large community of active users and developers, and the feedback from this community proved to be invaluable for the development of the OWL Plugin.

Our decision to base the OWL Plugin on Protégé also had some risks. In order to be able to reuse as much of the existing Protégé features as possible, we had to create a careful mapping between the Protégé metamodel and OWL that maintains the traditional Protégé semantics where possible. Furthermore, none of the generic Protégé widgets and tabs is optimized for OWL, and not all of the editing metaphors for frame-based systems are appropriate for OWL. In particular, OWL's rich description logics features such as logical class definitions required special attention. The following sections will show how we have addressed these issues.

4 The Architecture of the OWL Plugin

The OWL Plugin is a complex Protégé extension that can be used to edit OWL files and databases. The OWL Plugin includes a collection of custom-tailored tabs and widgets for OWL, and provides access to OWL-related services such as classification, consistency checking, and ontology testing.

4.1 OWL Plugin Metamodel

As illustrated in Figure 1, the OWL Plugin extends the Protégé model and its API with classes to represent the OWL specification. The OWL Plugin supports RDF(S), OWL Lite, OWL DL (except for anonymous global class axioms, which need to be given a name by the user) and significant parts of OWL Full (including metaclasses).

In order to better understand this extension mechanism, we need to look at the differences between the Protégé metamodel and OWL. OWL is an extension of RDF(S) [7]. RDF has a very simple triple-based model that is often too verbose to be edited directly in a tool. Fortunately, RDF Schema extends RDF with metamodel classes and properties which can be mapped into the Protégé metamodel. As a result, the extensions that OWL adds to RDF(S) can be reflected by extensions of the Protégé metamodel.

Although this extension has been successfully implemented for the OWL Plugin, not all aspects of the metamodels could be mapped trivially. It was straight-forward to represent those aspects of OWL that just extend the Protégé metamodel. For example, in order to represent disjoint class relationships, it was sufficient to add a new property `:OWL-DISJOINT-CLASSES` to Protégé's `owl:Class` metaclass. It was also relatively easy to represent OWL's complex class constructors that can build class descrip-

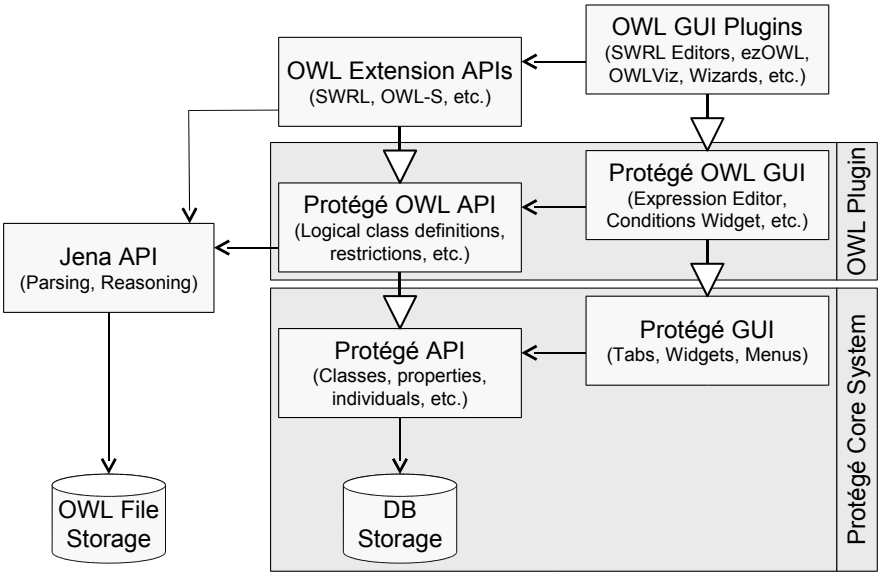

Fig. 1. The OWL Plugin is an extension of the Protégé core system.

tions out of logical statements. For example, OWL classes can be defined as the complement of other classes, using the `owl:complementOf` constructor. In the OWL Plugin, complements are represented by instances of a metaclass `:OWL-COMPLEMENT-CLASS` that inherits from other Protégé system classes. As illustrated in Figure 2, the other types of OWL class constructors such as restrictions and enumerated classes, and the various kinds of properties are mapped into similar metaclasses.

Other aspects of OWL required some work to maintain a maximum of backward compatibility with traditional Protégé applications. There is a semantic difference between Protégé and OWL if multiple restrictions are defined at the same time. In particular, Protégé properties with multiple classes as their range can take as values instances of all classes (union semantics), whereas OWL properties with multiple classes in their range can only take values that are instances of all classes at the same time (intersection semantics). In order to solve this mismatch, the OWL Plugin uses an internal `owl:unionOf` class if the user has defined more than one range class. The same applies to a property's domain. Another difference is that OWL does not have the notion of facets, which in Protégé are used to store property restrictions at a class. While a maximum cardinality restriction at a class in Protégé is represented by a single quadruple (class, property, facet, value), the same is stored as an anonymous superclass in OWL. OWL even supports attaching annotation property values to such anonymous classes, and therefore it would be insufficient to map OWL restrictions into facets only. We have implemented a mechanism that automatically synchronizes facet values with restriction classes, so that the traditional semantics of Protégé are maintained while using the syntax of OWL.

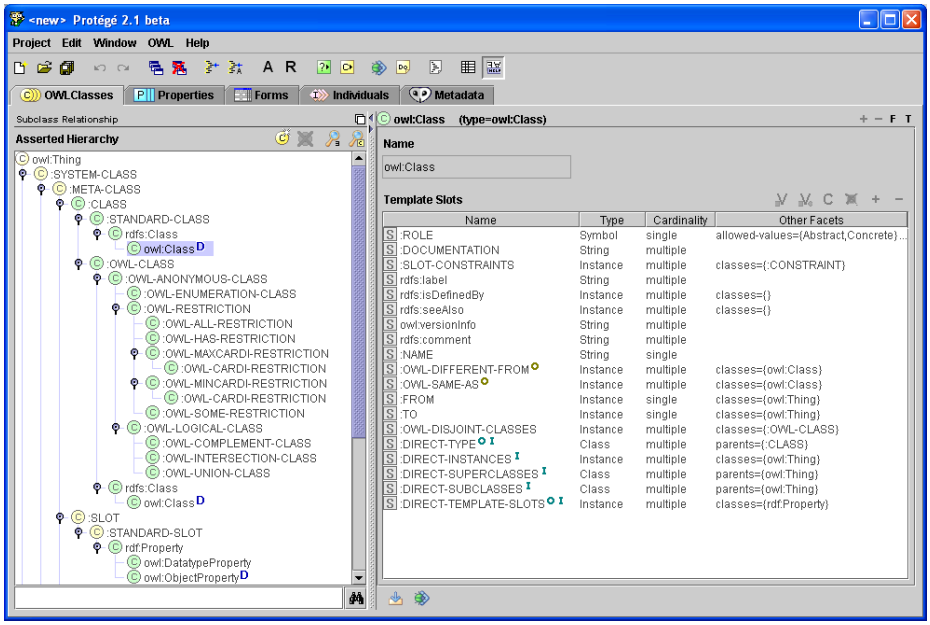

Fig. 2. The OWL metaclasses are implemented as subclasses of the Protégé system classes. As shown here, they can be browsed with Protégé as well.

4.2 OWL Plugin API

Reflecting the Protégé metamodel extensions, the OWL Plugin also provides an extended Java API to access and manipulate OWL ontologies. While the core API already provides access to ontology classes, properties, and instances, the OWL Plugin extends this API with custom-tailored Java classes for the various OWL class types. This API basically encapsulates the internal mapping and thus shields the user from error-prone low-level access. It is possible to further extend this API to define custom-tailored classes for OWL extensions like OWL-S and the Semantic Web Rule Language (SWRL) [6]. For example, individuals of the SWRL class `AtomList` could be represented by instances of a corresponding Java class such as `AtomListInstance`. Algorithms for SWRL could then operate on more convenient objects than with the generic classes, while non-SWRL-aware Protégé components would handle these objects as normal `Instances`. We are currently working such a SWRL library.

The OWL Plugin provides a comprehensive mapping between its extended API and the standard OWL parsing library Jena [1]. After an ontology has been loaded into a Jena model, the OWL Plugin generates the corresponding Protégé objects. It then keeps the Jena model in memory at all times, and synchronizes it with all changes performed by the user. Thus, if the user creates a new Protégé class, a new Jena class with the same name is created as well. The presence of a secondary representation of the ontology in

[1] `http://jena.sourceforge.net`

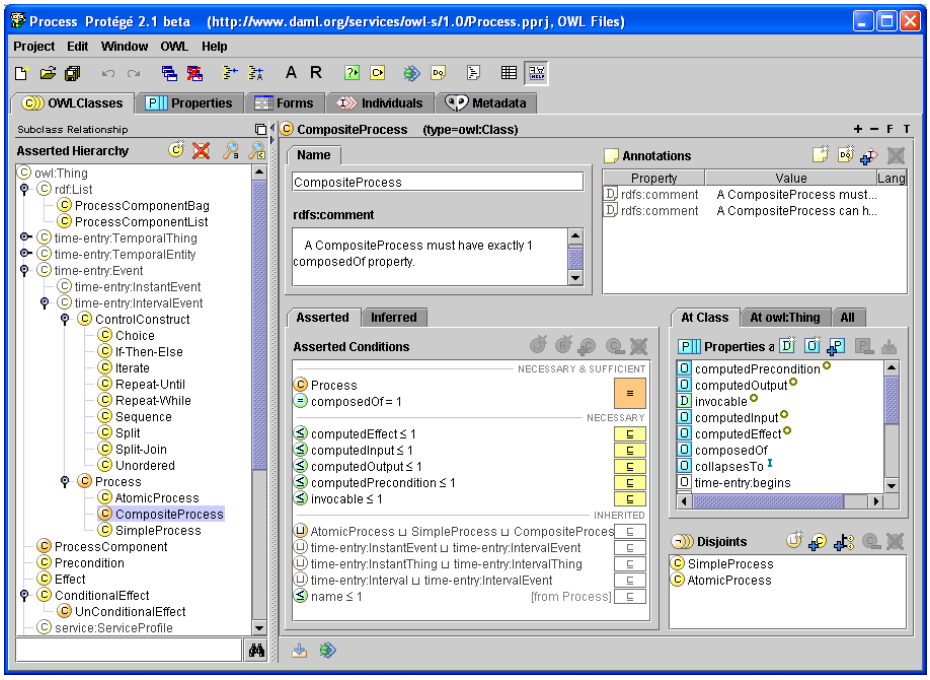

Fig. 3. A screenshot of the classes tab in the OWL Plugin (displaying the OWL-S ontology). The screenshot shows the logical definition of the selected class `CompositeProcess`, its properties, disjoints and annotations.

terms of Jena objects means that the user is able to invoke arbitrary Jena-based services such as interfaces to classifiers, query languages, or visualization tools permanently. The mapping into Jena also makes it much easier to embed existing and future Semantic Web services into the OWL Plugin. Also, when the ontology is saved, it is sufficient to call the corresponding Jena API method to serialize the Jena objects to a file. The immediate Jena mapping is not available in the OWL Plugin's database or multi-user modes, where it is replaced with a conventional, monolithic translation.

4.3 OWL Plugin User Interface

Based on the above mentioned metamodel and API extensions, the OWL Plugin provides several custom-tailored graphical user interface components for OWL. When started, the system displays the five tabs shown in Figure 3. Most ontology designers will focus on the *OWL classes* and *Properties* tabs which are described in sections 5 and 6. The *Forms* and *Individuals* tabs are mostly geared for the acquisition of Semantic Web contents (instance data, detailed in Section 7), while the *Metadata* tab allows users to specify global ontology settings such as imports and namespaces.

Note that the generic architecture of Protégé and the OWL-specific extensions make it relatively easy to add custom-tailored components. For example, optimized editors for SWRL or OWL-S could be added to the system. Likewise, description-logic reasoners could be directly implemented on top of the Protégé OWL API or Jena. In fact, several developers from around the world have already developed extensions of the OWL Plugin, some of them even without interacting with us. In the remainder of this document, we will focus on the standard set of user interface components and features of the OWL Plugin. Since all of these components are available as open-source, it is possible to extend and customize them.

5 Editing OWL Ontologies

In this section we will walk through some of the forms and tabs for editing classes and properties. Further details on the user interface (including a comparison with similar tools such as OilEd) can be found in complementary publications [10, 9].

A screenshot of the OWL classes tab is shown in Figure 3. The main class hierarchy is shown on the left, and the details of the currently selected class are shown in a form on the right. The upper section of the class form displays class metadata such as names and annotations. Annotation properties are ignored by OWL DL reasoners. Instead, they are very suitable to manage metadata about a class, such as versioning information, comments, relationships to other external resources, and labels in multiple languages.

5.1 Displaying and Editing OWL Expressions

A key question for developers of an OWL tool is how to represent and edit logical class descriptions in a way that makes them easy to read and, at the same time, efficient to enter. With OWL's RDF-based syntax [16], expressions quickly become extremely verbose and hard to read. The OWL Abstract Syntax [17] is much more user-friendly, but still quite verbose. Although Protégé also has some support for the Abstract Syntax, we chose to develop an expression syntax based on standard DL symbols [1] such as $\forall$ and $\sqcup$ as the primary display format. These symbols (Table 1) allow the system to display complex nested expressions in a single row.

This notation is used consistently throughout the user interface and is supported by a comfortable expression editor. Using this editor, users can rapidly enter OWL class expressions either with mouse or keyboard. The special characters are mapped onto keys known from languages such as Java (e.g., `owl:intersectionOf` is entered with the `&` key). To simplify editing, keyboard users can exploit a syntax completion mechanism known from programming environments, which semi-automatically completes partial names after the user has pressed `tab`. For very complex expressions, users can open a multi-line editor in an extra window, which displays the expression using indentation.

The OWL Plugin helps new users to get acquainted with the expression syntax. English prose text is shown as a "tool tip" when the mouse is moved over the expression. For example, "$\exists$ *hasPet* `Cat`" is displayed as "Any object which has a cat as its pet".

OWL element	Symbol	Key	Example expression in Protégé
owl:allValuesFrom	∀	*	∀ *hasChildren* Female
owl:someValuesFrom	∃	?	∃ *hasHabitat* University
owl:hasValue	∋	$	*hasGender* ∋ male
owl:minCardinality	≥	>	*hasChildren* ≥ 1 (at least one value)
owl:maxCardinality	≤	<	*hasDegree* ≤ 5 (at most five values)
owl:cardinality	=	=	*hasGender* = 1 (exactly one value)
owl:intersectionOf	⊓	&	Student ⊓ Parent
owl:unionOf	⊔	|	Male ⊔ Female
owl:complementOf	¬	!	¬Parent
owl:oneOf	{ ... }	{ }	{yellow green red}

Table 1. Protégé uses traditional description logic symbols to display OWL expressions. In this table, property names such as *hasChildren* appear in italics. A common naming convention is to use uppercase names such as Parent to represent classes, while individuals like yellow should be written in lower case.

5.2 Editing Class Descriptions

Traditional Protégé users are accustomed to an object-centered view to the interface that has required some effort to adapt to OWL. In the Protégé metamodel, classes are typically only related through simple superclass/subclass relationships, and therefore a simple tree view was enough to edit classes. OWL on the other hand not only distinguishes between necessary conditions (superclasses) and necessary and sufficient conditions (equivalent classes), but furthermore allows users to relate classes with arbitrary class expressions. As shown in Figure 3, the OWL Plugin's class editor addresses this complexity by means of a list of conditions, which is organized into blocks of necessary & sufficient, necessary, and inherited conditions. Each of the necessary & sufficient blocks represents a single equivalent intersection class, and only those inherited conditions are listed that have not been further restricted higher up in the hierarchy. In addition to the list of conditions, there is also a custom-tailored widget for entering disjoint classes, which has special support for typical design patterns such as making all siblings disjoint. This rather object-centered design of the OWL classes tab makes it possible to maintain the whole class definition on a single screen.

5.3 Editing Properties

The class form provides a listing of all properties for which instances of the selected class can take values. This includes those properties which have the class in their domain, and those that don't have any domain restrictions. The details of the properties are edited by means of a separate form. Similar to the class form, the upper part of the property form displays name and annotation properties. The lower part contains widgets for the property's domain, range, and characteristics such as whether a property is transitive or symmetric.

Note that Protégé and OWL support user-defined metaclasses that extend the standard system classes. For example, ontologies can define a subclass of owl:Object-Property and then add other properties to it. This allows users to specify additional

property characteristics such as *hasUnit*. The system automatically creates widgets for the additional properties of the metaclass.

6 Ontology Maintenance and Evolution

As mentioned in Section 2, ontology design is a highly evolutionary process. Ontology developers almost certainly will need to explore various iterations before an ontology can be considered to be complete. A development tool should assist in ontology evolution, and (where appropriate) help the user to prevent or circumnavigate common design mistakes. In the OWL Plugin, we are exploring some promising approaches for ontology maintenance, partly comparable to modern tools for programming languages. With programming tools, developers can get instant feedback using the *compile* button. Compiler errors are listed below the source code and enable the programmer to quickly navigate to the affected area. Another very efficient means of detecting programming errors is using so-called *test cases*, which have become popular in conjunction with agile development approaches such as Extreme Programming [2]. A test case is a small piece of code that simulates a certain scenario and then tests whether the program behaves as expected. It is a good programming style to maintain a library of test cases together with the source code, and to execute all test cases from time to time to verify that none of the recent changes has broken existing functionality. To a certain extent, the idea of test cases is related to the formal class definitions in description logics such as OWL DL. For example, by formally stating that a `Parent` is the intersection of `Person` and a minimum cardinality restriction on the *hasChildren* property we ensure that future statements about `Parents` don't contradict the original developer's intention. This is especially important in an open-world scenario such as the Semantic Web. Thus, DL reasoners can help build and maintain sharable ontologies by revealing inconsistencies, hidden dependencies, redundancies, and misclassifications [14]. How the OWL Plugin integrates such reasoners is illustrated in Section 6.1. In addition to reasoners, the OWL Plugin also adopts the notions of test cases and compile buttons with an "ontology testing" feature, which is described in Section 6.2.

6.1 Reasoning based on Description Logics

The OWL Plugin provides direct access to DL reasoners such as Racer [5]. The current user interface supports two types of DL reasoning: Consistency checking and classification (subsumption). Support for other types of reasoning, such as instance checking, is work in progress.

Consistency checking (i.e., the test whether a class could have instances) can be invoked either for all classes with a single mouse click, or for selected classes only. Inconsistent classes are marked with a red bordered icon.

Classification (i.e., inferring a new subsumption tree from the asserted definitions) can be invoked with the classify button on a one-shot basis. When the classify button is pressed, the system determines the OWL species, because some reasoners are unable to handle OWL Full ontologies. This is done using the validation service from the Jena library. If the ontology is in OWL Full (e.g., because metaclasses are used) the system

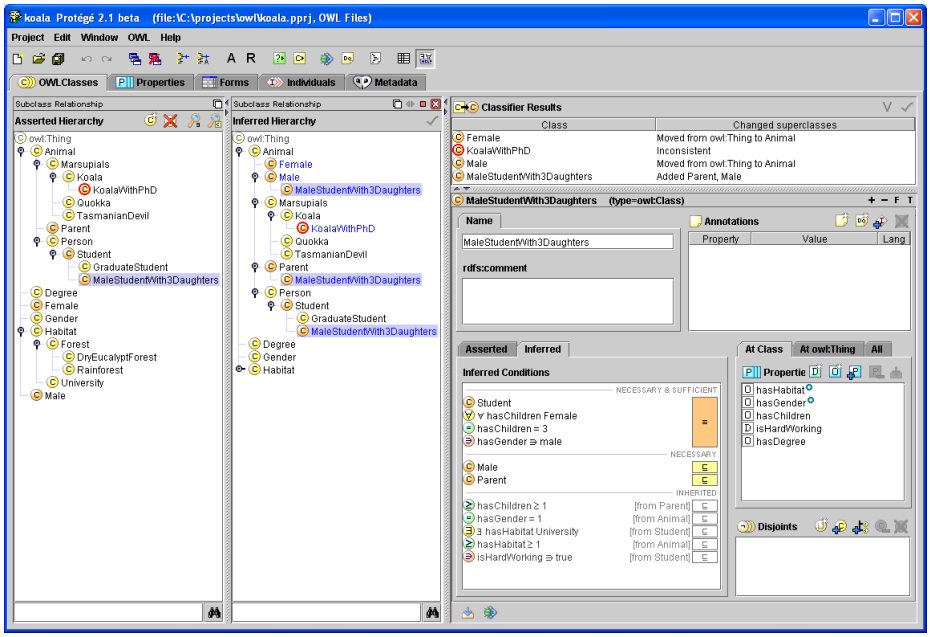

Fig. 4. The OWL Plugin can be used to invoke a classifier and to visualize classification results in support of ontology maintenance. The asserted and the inferred hierarchy are shown side by side.

attempts to convert the ontology temporarily into OWL DL. The OWL Plugin supports editing some features of OWL Full (e.g., assigning ranges to annotation properties, and creating metaclasses). These are easily detected and can be removed before the data are sent to the classifier. Once the ontology has been converted into OWL DL, a full consistency check is performed, because inconsistent classes cannot be classified correctly. Finally, the classification results are stored until the next invocation of the classifier, and can be browsed separately. Classification can be invoked either for the whole ontology, or for selected subtrees only. In the latter case, the transitive closure of all accessible classes is sent to the classifier. This may return an incomplete classification because it does not take incoming edges into account, but in many cases it provides a reasonable approximation without having to process the whole ontology.

OWL files store only the subsumptions that have been asserted by the user. However, experience has shown that, in order to edit and correct their ontologies, users need to distinguish between what they have asserted and what the classifier has inferred. Many users may find it more natural to navigate the inferred hierarchy, because it displays the semantically correct position of all the classes.

The OWL Plugin addresses this need by displaying both hierarchies and making available extensive information on the inferences made during classification. As illustrated in Figure 4, after classification the OWL Plugin displays an inferred classification hierarchy beside the original asserted hierarchy. The classes that have changed

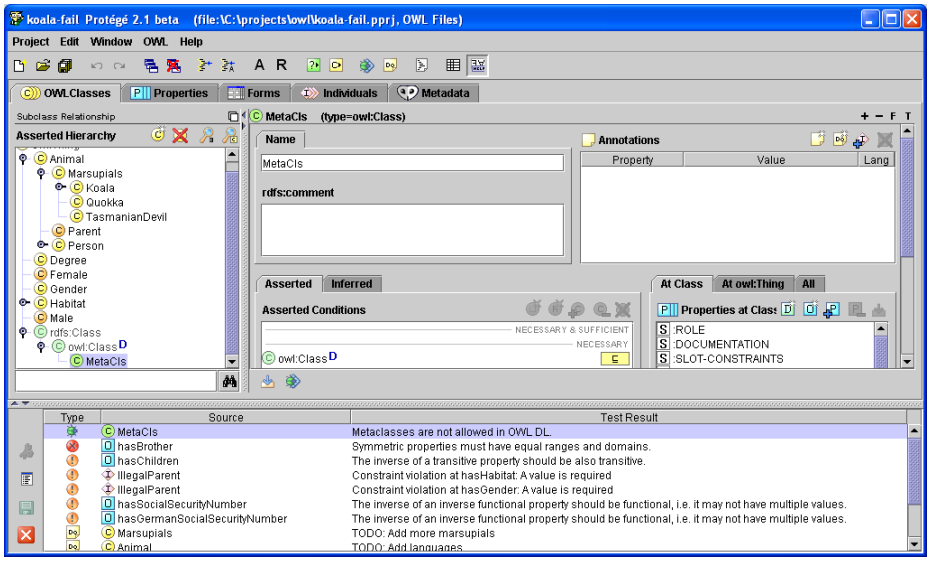

Fig. 5. The OWL Plugin displays violations of test conditions at the bottom.

their superclasses are highlighted in blue, and moving the mouse over them explains the changes. Furthermore, a complete list of all changes suggested by the classifier is shown in the upper right area, similar to a list of compiler messages. A click on an entry navigates to the affected class. Also, the conditions widget can be switched between asserted and inferred conditions. All this allows the users to analyze the changes quickly.

6.2 Ontology Testing

The OWL Plugin provides a mechanism to execute small test cases. Users can press an ontology test button, and the system will execute a configurable list of tests. These tests are small Java programs that basically take a class, property, individual, or ontology as its input, verify arbitrary conditions on them, and in case of failure, return an error message. For example, one of the predefined tests ensures the invariant that the inverse of a transitive property should also be transitive. As illustrated in Figure 5, if a property in the ontology violates this conditions, then the system displays a warning. In some cases it even provides a “repair” button, which attempts to remove the source of the violation automatically.

The OWL Plugin provides a standard set of ontology tests for various best ontology design practices. It also contains tests against OWL DL compliance (e.g., to warn the user when OWL Full features such as metaclasses have been used). The ontology test mechanism has also been exploited to implement a simple but powerful “to-do-list” feature. For example, if a class has the value “TODO: Add German label” as its value of the `owl:versionInfo` annotation property, then the ontology test button will

display a corresponding to-do item in the list of warnings. This simple mechanism helps coordinate shared ontology design efforts.

The list of standard ontology tests can be easily extended by programmers, so that the system will execute additional user-defined tests uniformly. These additional tests could for example ensure the application of project-specific design patterns, naming conventions, or other best practices.

7 Editing Semantic Web Contents

Ontologies provide Semantic Web agents with background knowledge about domain concepts and their relationships. This knowledge can be exploited in various ways, for example to drive context-sensitive search functions. Ontologies can also be instantiated to create individuals that describe Semantic Web resources or real-world entities. For example, individuals of an ontology for travel agents could represent specific holiday destinations or activities. In such a scenario, a Semantic Web repository would provide instance data about these individuals, and agents can use their ontological knowledge to match user requests with available offers.

Protégé provides out-of-the-box support for editing individuals. The OWL Plugin's *Individuals* tab can be used to instantiate classes from the ontology, and to edit the characteristics of the individuals with comfortable forms. Users could import an ontology into their project, create instances for its classes, and then store the instances into separate files. These files can then be distributed on the Web, so that intelligent agents could find them.

In another application scenario, Protégé could be used to associate existing Web resources such as images to ontological concepts. In the context of another Protégé plugin, we have implemented a convenient drag-and-drop mechanism that allows users to drag a Web link or image from their Web browser into Protégé, and thus establish a relationship between the external Web resource and the selected object in the ontology. These relationships could be exploited by arbitrary Web Services. Thus, Protégé is not only limited to being used as an ontology authoring tool, but also as a platform for arbitrary other services that *use* the ontologies.

8 Discussion

While real Semantic Web applications are still in their infancy, there is a clear demand for tools that assist in application development. The intention of the Protégé OWL Plugin is to make Semantic Web technology available to a broad group of developers and users, and to promote best practices and design patterns.

One of the major benefits of using Protégé is its open architecture. The system provides various mechanisms to hook custom-tailored extensions into it, so that external components like reasoners and Web services can be integrated easily. Since the source code is open and freely available as well, existing base components can be used as templates for customized solutions. Projects don't need to spend time developing their own

base infrastructure with standard features such as loading and managing OWL ontologies. Instead, they can start with the OWL Plugin as it comes out-of-the-box and then gradually adapt or remove the features that don't completely match their requirements.

Since its first beta versions in late 2003, the OWL Plugin has been widely embraced by OWL users around the world. Although we don't have exact numbers and statistics about our users, we know that Protégé has more than 20,000 registered users. Of these, a significant portion is very interested in OWL and many use it routinely. The `protege-owl@smi.stanford.edu` discussion list currently has more than 650 subscribed members. The traffic on this list is very large and provides useful critiques for the developers, as well as encouraging feedback and success stories. There is already a considerable number of external plugins for the OWL Plugin, demonstrating that the open architecture of Protégé provides a suitable platform for custom extensions. Also, many existing Protégé Plugins either directly work in OWL mode, or are being optimized for OWL.

The decision to implement the OWL Plugin as an extension to Protégé did not have only advantages though. In particular, the Protégé core metamodel and API support some functionality that does not have a default mapping into OWL, such as numeric range restrictions, assigning duplicate values to a property, or stating that a class is abstract (i.e., cannot have any instances). As a result, some existing widgets could not be used in the OWL mode, and programmers should not use all low-level API functions. We have made some efforts to simulate some Protégé-specific language elements with OWL. For example, if users want to state that a class is abstract, then the system fills a pre-defined annotation property. Furthermore, invalid API calls are rejected whenever possible. We believe that the many advantages of reusing the functionality and components of the Protégé platform clearly outweigh these inconveniences.

Another point to keep in mind is that the existing collection of user interface components in the OWL Plugin should only be regarded as a core configuration. For example, many users may be put off by the symbolic notation used to edit class expressions. These users often don't require the advanced description logic features of OWL but simply want to define classes and properties similar to the conventional Protégé user interface or object-oriented tools. These users may find it helpful to simplify their user interface. This would allow them to use the advanced features later, once they are accustomed to the basics of OWL. We have recently added an alternative default mode of the OWL Classes tab (called "Properties View"), which shows properties together with their restrictions instead of the generic conditions widget. This view is closer to the view that traditional Protégé users are accustomed to, but has some limitations on the full expressivity of OWL.

The focus of our work on the OWL Plugin until now has been to provide a complete, scalable and stable editor for most of the features of OWL (Full). Ongoing and future work addresses the question of how to make ontology design more accessible to people with limited formal training in description logics. In this context, we collaborate with the University of Manchester to explore more intuitive visual editing components, wizards, and debugging aids. We will also support additional design patterns, improve reasoning capabilities, and build editors and APIs for OWL extensions such as SWRL.

Acknowledgements. This work has been funded by a contract from the US National Cancer Institute and by grant P41LM007885 from the National Library of Medicine. Additional support for this work came from the UK Joint Information Services Committee under the CO-ODE grant. Our partners from Alan Rector's team and the OilEd developers at the University of Manchester have made very valuable contributions.

References

1. F. Baader, D. Calvanese, D. McGuineness, D. Nardi, and P. Patel-Schneider, editors. *The Description Logic Handbook*. Cambridge University Press, 2003.
2. K. Beck. *Extreme Programming Explained: Embrace Change*. Addison-Wesley, 1999.
3. T. Berners-Lee, J. Hendler, and O. Lassila. The Semantic Web. *Scientific American*, 284(5):34–43, 2001.
4. J. Gennari, M. Musen, R. Fergerson, W. Grosso, M. Crubézy, H. Eriksson, N. Noy, and S. Tu. The evolution of Protégé-2000: An environment for knowledge-based systems development. *International Journal of Human-Computer Studies*, 58(1):89–123, 2003.
5. V. Haarslev and R. Moeller. Racer: A core inference engine for the Semantic Web. In *2nd International Workshop on Evaluation of Ontology-based Tools (EON-2003)*, Sanibel Island, FL, 2003.
6. I. Horrocks and P. F. Patel-Schneider. A proposal for an OWL rules language. In *Proc. of the Thirteenth International World Wide Web Conference (WWW 2004)*, New York City, NY.
7. I. Horrocks, P. F. Patel-Schneider, and F. van Harmelen. From SHIQ and RDF to OWL: The making of a web ontology language. *Journal of Web Semantics*, 1(1), 2003.
8. H. Knublauch. An AI tool for the real world: Knowledge modeling with Protégé. *JavaWorld*, June 20, 2003.
9. H. Knublauch, O. Dameron, and M. A. Musen. Weaving the biomedical semantic web with the Protégé OWL plugin. In *International Workshop on Formal Biomedical Knowledge Representation*, Whistler, BC, Canada, 2004.
10. H. Knublauch, M. A. Musen, and A. L. Rector. Editing description logics ontologies with the Protégé OWL plugin. In *International Workshop on Description Logics*, Whistler, BC, Canada, 2004.
11. N. Noy, R. Fergerson, and M. Musen. The knowledge model of Protégé-2000: Combining interoperability and flexibility. In *2nd International Conference on Knowledge Engineering and Knowledge Management (EKAW'2000)*, Juan-les-Pins, France, 2000.
12. N. Noy, M. Sintek, S. Decker, M. Crubézy, R. Fergerson, and M. Musen. Creating Semantic Web contents with Protégé-2000. *IEEE Intelligent Systems*, 2(16):60–71, 2001.
13. N. F. Noy and D. L. McGuinness. Ontology development 101: A guide to creating your first ontology. Technical Report SMI-2001-0880, Stanford Medical Informatics, 2001.
14. A. L. Rector. Description logics in medical informatics. Chapter in [1].
15. A. L. Rector. Modularisation of domain ontologies implemented in description logics and related formalisms including OWL. In *Second International Conference on Knowledge Capture (K-CAP)*, Sanibel Island, FL, 2003.
16. World Wide Web Consortium. OWL Web Ontology Language Reference. W3C Recommendation 10 Feb, 2004.
17. World Wide Web Consortium. OWL Web Ontology Language Semantics and Abstract Syntax. W3C Recommendation 10 Feb, 2004.