NeOn: Lifecycle Support for Networked Ontologies

Integrated Project (IST-2005-027595)

Priority: IST-2004-2.4.7 — "Semantic-based knowledge and content systems"

D1.5.3 Advanced Methods for Change Propagation between Networked Ontologies and Metadata

Deliverable Co-ordinator: **Diana Maynard**

Deliverable Co-ordinating Institution: **University of Sheffield (USFD)**

Other Authors: **Niraj Aswani (USFD); Wim Peters (USFD); Fouad Zablith (OU); Mathieu d'Aquin (OU)**

This document describes the implementation of the approach to modelling some of the dynamics of (semantic) metadata that were described in D1.5.1 [MPD+07] and D1.5.2 [MPAD08]. We focus here specifically on the interaction between the example clients and the NeOn toolkit, such that changes in networked ontologies can be propagated to the semantic metadata, and vice versa. This means that, for example, changes made to an ontology using GATE services can be propagated to the NeOn toolkit where other users can access them, and that changes made to ontologies by other users via the toolkit can be accessed by users of GATE services. The deliverable comprises two main parts. First, we describe the implementation of ontology change between GATE and the NeOn Toolkit, and second, we describe the Evolva plugin which enables ontological changes to be captured and propagated to the toolkit from folksonomy data.

Document Identifier:	NEON/2009/D1.5.3/v1.0	Date due:	February 28, 2009
Class Deliverable:	NEON EU-IST-2005-027595	Submission date:	February 28, 2009
Project start date	March 1, 2006	Version:	v1.0
Project duration:	4 years	State:	Final
		Distribution:	Public

Work package participants

The following partners have taken an active part in the work leading to the elaboration of this document, even if they might not have directly contributed to the writing of this document or its parts:

- University of Sheffield
- Open University
- University of Karlsruhe

Change Log

Version	Date	Amended by	Changes
0.1	12-01-2009	Diana Maynard	First template
0.2	29-01-2009	Diana Maynard	First version sent for QA
0.3	19-01-2009	Niraj Aswani	Edits after QA comments
0.4	23-01-2009	Fouad Zablith	Edits after QA comments
0.5	23-01-2009	Diana Maynard	Final version

Executive Summary

This document describes the implementation of the approach to modelling some of the dynamics of (semantic) metadata that were described in D1.5.1 [MPD+07] and D1.5.2 [MPAD08]. We focus here specifically on the interaction between the example clients and the NeOn toolkit, such that changes in networked ontologies can be propagated to the semantic metadata, and vice versa. This means that, for example, changes made to an ontology using GATE services can be propagated to the NeOn toolkit where other users can access them, and that changes made to ontologies by other users via the toolkit can be accessed by users of GATE services. The deliverable comprises two main parts. First, we describe the implementation of ontology change between GATE and the NeOn Toolkit, and second, we describe the Evolva plugin which enables ontological changes to be captured and propagated to the toolkit from folksonomy data.

The objectives of this deliverable are twofold:

1. to achieve interoperability of NeOn Toolkit and GATE ontology change management at the change log level, in close cooperation with T1.3, and to finalise the implementation of two-way change propagation, such that ontology changes in GATE can be propagated to the NeOn Toolkit and vice versa;

2. the development of an ontology evolution plugin for the NeOn Toolkit which enables ontologies in the toolkit to be modified and extended by means of semantic metadata found in textual documents and folksonomies.

Contents

List of Figures

Chapter 1

Introduction

Ontology evolution is increasingly getting research momentum in the Semantic Web field. This is due to the fact that ontologies, forming the backbone of Semantic Web systems, need to be kept up-to-date so that ontology-based systems remain usable. In D1.5.2, we described the implementation of some methodologies for dealing with ontology and metadata change in text. These are motivated by the need to propagate changes in an ontology to its instances and properties, and vice versa. For example, if a user deletes concepts from the ontology, it is important to have a mechanism for dealing with any associated semantic metadata, in order not to lose vital information. If a user adds new concepts to the ontology, then it may be necessary to return to the text to check whether additional instances can be found which should be used to populate these new concepts in the ontology. We call this the top-down approach to ontology change. Furthermore, not only are ontologies dynamic and subject to structural change, but so are the texts and instances from which the ontologies may be derived. If we get additional relevant textual material and/or find new instances in that text, it may be necessary to modify the ontology to take into consideration this new information (for example, adding new concepts or new relations between existing concepts in the ontology). We call this the bottom-up approach to ontology change.

Our top-down approach enables changes made to the ontology to be propagated to the metadata so that as little information as possible is lost. For example, when a concept is deleted from the ontology, usually any semantic metadata (instances) belonging to that concept would be deleted with it, but this is not always desirable behaviour because often we would prefer to reclassify the instance at a more general level. We therefore created a methodology for change propagation (as described in D1.5.1), on which the implementation of our ontology change typology was based (as described in D1.5.2).

However, while the resulting implementation enabled a user to be aware of changes to the ontology made by another person at a distributed location (thereby aiding collaborative annotation), this was only possible if all such changes were made in GATE. Changes made to an ontology in the NeOn toolkit could not be propagated to the GATE ontology management system, and vice versa. In the first part of this deliverable, we describe the work undertaken to rectify this problem, by linking the change log created by GATE to the change log in the NeOn toolkit, such that changes can be propagated in both directions. This is very important for collaborative distributed ontology management.

In D1.5.2 we also described methodologies for capturing changes to ontologies based on data from folksonomies and from textual documents (bottom-up approach). In this deliverable, we describe the practical implementation of this work in the form of Evolva, an ontology evolution system that can extend an ontology based on information extracted from various data sources. It supports both textual sources, from which it extracts representative terms, and folksonomies, from which it extracts tags which frequently co-occur with a given concept. The extracted tags and terms are then used to extend the given ontology. In order to perform this linking, various background knowledge sources are used such as WordNet [Fel98] and the Semantic Web.

The objectives of this deliverable are thus twofold:

1. to achieve interoperability of NeOn Toolkit and GATE ontology change management at the change log

level, in close cooperation with T1.3, and to finalise the implementation of two-way change propagation, such that ontology changes in GATE can be propagated to the NeOn Toolkit and vice versa;

2. the development of an ontology evolution plugin for the NeOn Toolkit which enables ontologies in the toolkit to be modified and extended by means of semantic metadata found in textual documents and folksonomies.

1.1 Related Deliverables

The work in this deliverable is strongly related to a number of other deliverables in NeOn:

1. It describes the implementation of the methods for ontology change management proposed in D1.5.1 and D1.5.2;

2. It is closely related to the work on change management to support collaborative workflows described in D1.3.2. The implementation of our change log system in GATE is designed specifically to interact with the change log and workflows implemented there.

3. The bottom-up approaches to ontology change management such as Evolva, SPRAT and SARDINE are closely related to work carried out in WP2 and WP7. SPRAT is described in more detail in D2.2.2. All these approaches are also designed to interact with the change log system implemented.

Chapter 2

GATE Change log

2.1 Introduction

In previous deliverables [PHWD08, PHJ09] we described an ontology change log management system developed for the NeOn Toolkit which allows the managing of changes made by users in a collaborative environment. This is necessary because different people, who may be experts in their domain, sometimes end up making changes to the same ontology, often at distributed locations. Where it is not possible to have access to a shared repository, other people wishing to make changes to the same ontology have to wait for others to finish their tasks. Changes made to an ontology can involve additions, deletions and modifications (see [MPD$^+$07] for more details).

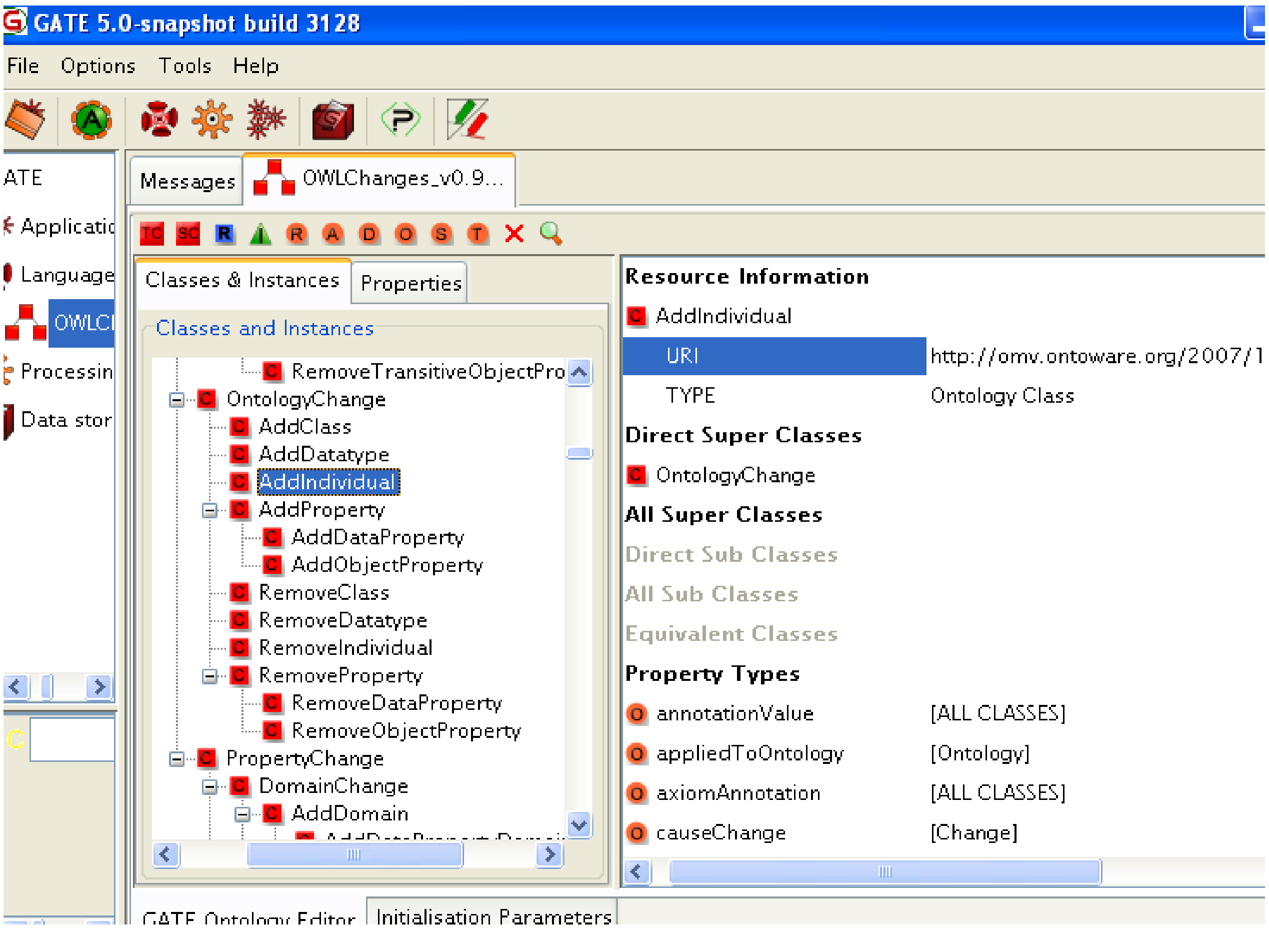
The basic idea behind producing a change log is to allow people to share their changes with others and thus collaborate on manipulating ontology data. This essentially allows them to work independently as well as simultaneously on the same ontology. However, some ontologies can be extremely large and unwieldy, whereas the changes made to such ontologies may be only very minor. Instead of exchanging different versions of ontologies, it makes it easier to exchange change logs which can then be applied over to the original ontology to get the modified version.

As an extension to the OMV[1], a taxonomy has been defined called *OWLChanges*, consisting of a concept in the ontology representing each different type of change possible. Figure 2.1 shows a snapshot of part of this ontology, loaded in GATE. This includes concepts such as *AddClass*, *RemoveClass*, *AddIndividual*, *RemoveIndividual* etc. In the case of the NeOn Toolkit change log, every change made to an ontology is recorded as an instance of the relevant concept in the *OWLChanges* ontology. Recording changes this way makes it possible to carry the final change log as a separate ontology where every instance refers to a change made in the ontology. The change log, in addition to the information about changes made by users, contains other useful information such as who made those changes, when and in which order etc. Since the change log is a valid OWL ontology, it can be then loaded as well as queried independently.

GATE has several different plugins useful for carrying out different types of information extraction tasks. It also has its own ontology API, which can be used together with these resources to take the maximum advantage of the combination. There are several applications that use these resources and manipulate ontologies (automatically) using the ontology API. We implemented GATE Ontology Services to allow people to connect to a central repository and manage ontologies centrally. It has been implemented in such way that it can be used with other resources available in GATE. In D1.5.2 [MPAD08], we described a plugin called the *NeonOntologyServiceClient* that allows connecting to the GATE Ontology Services (GOS).

GOS supports storing ontologies on a shared repository on a remote server. GOS is based on OWLIM which is a high performance semantic repository developed in Java. It is packaged as a Storage and Inference layer (SAIL) for the Sesame[2] RDF database. It has its own published API that a user can use to make changes to the ontologies stored on the server. Software clients such as the *NeonOntologyServiceClient* allow users

[1] http://ontoware.org/projects/omv/
[2] http://www.openrdf.org/



Classes and Instances

- RemoveTransitiveObjectPro
- OntologyChange
  - AddClass
  - AddDatatype
  - AddIndividual
  - AddProperty
    - AddDataProperty
    - AddObjectProperty
  - RemoveClass
  - RemoveDatatype
  - RemoveIndividual
  - RemoveProperty
    - RemoveDataProperty
    - RemoveObjectProperty
- PropertyChange
  - DomainChange
    - AddDomain

Resource Information

AddIndividual

URI	http://omv.ontoware.org/2007/1
TYPE	Ontology Class

Direct Super Classes

OntologyChange

All Super Classes

Direct Sub Classes

All Sub Classes

Equivalent Classes

Property Types

annotationValue	[ALL CLASSES]
appliedToOntology	[Ontology]
axiomAnnotation	[ALL CLASSES]
causeChange	[Change]

Figure 2.1: Part of the OWLChanges ontology loaded in GATE

to connect to this server, upload new ontologies or use existing ones and manipulate data using the service methods published by the GOS. Similar to the NeOn Toolkit, GOS also produces a change log that describes changes made by the various users.

When making changes to the ontology, users contribute to the change log maintained on the server. The GOS maintains only one instance of change log per repository and accumulates changes made by different users in the same change log. Users wishing to download the entire change log can do so by selecting one of the options provided in the *NeonOntologyServiceClient*. When users download a change log, it not only contains changes made by them but the changes that took place ever since the server was started.

In its current version, GOS inherits Sesame's repository management system which allows giving different rights to different users on different repositories. In other words if proper rights are assigned, multiple users can connect to the same repository from different locations and make changes. It is up to the system using OWLIM/Sesame as backend to utilise this functionality and restrict multiple users from editing the same ontology at the same time.

In GOS, currenly, only one user is allowed to modify the ontology at a time. Other users are given read only access if they connect to a repository which is already being edited by another user. However, since the different repositories are stored on the same central server, different users connecting to the same repository one after the other, see the latest modified version. If simultaneous access is needed, one can carefully distribute or make copies of the same ontology across different repositories and ask users to connect to a different copy. Users wishing to make changes can then contribute by adding new instances or deleting existing ones in their copy. GOS does not allow making changes to the implicit resources (imported ontologies).

This helps in restricting users from making changes to the basic taxonomy. Since making changes produces change logs, these change logs can be investigated for debugging purposes. More information on GOS, how to access it, its methods and the change log can be found in D1.5.2 [MPAD08].

One of the problems with the GOS change log described previously is that it restricts users to use it only within GOS and is not directly compatible with the NeOn Toolkit. Change logs produced by GOS could be interpreted by GOS only. The same is true for NeOn Toolkit change logs which could not be interpreted by GOS. This meant that users of GATE who modify an ontology could not publish their changes to the NeOn Toolkit so that toolkit users having access to the same ontology could make use of them. Similarly if a toolkit user modified an ontology, the change log was not applicable in GATE.

We have tried to solve this problem through this deliverable. What we provide is an implementation in the GOS that not only understands but also produces the changelogs that are compatible with NeOn Toolkil. In other words, it bridges the gap between the GOS and the NeOn Toolkil that will allow people to carry changelogs (instead of the entire ontologies) across the two systems. Figure 2.2 shows the interaction between GATE application, the change log created and the Neon Toolkit.

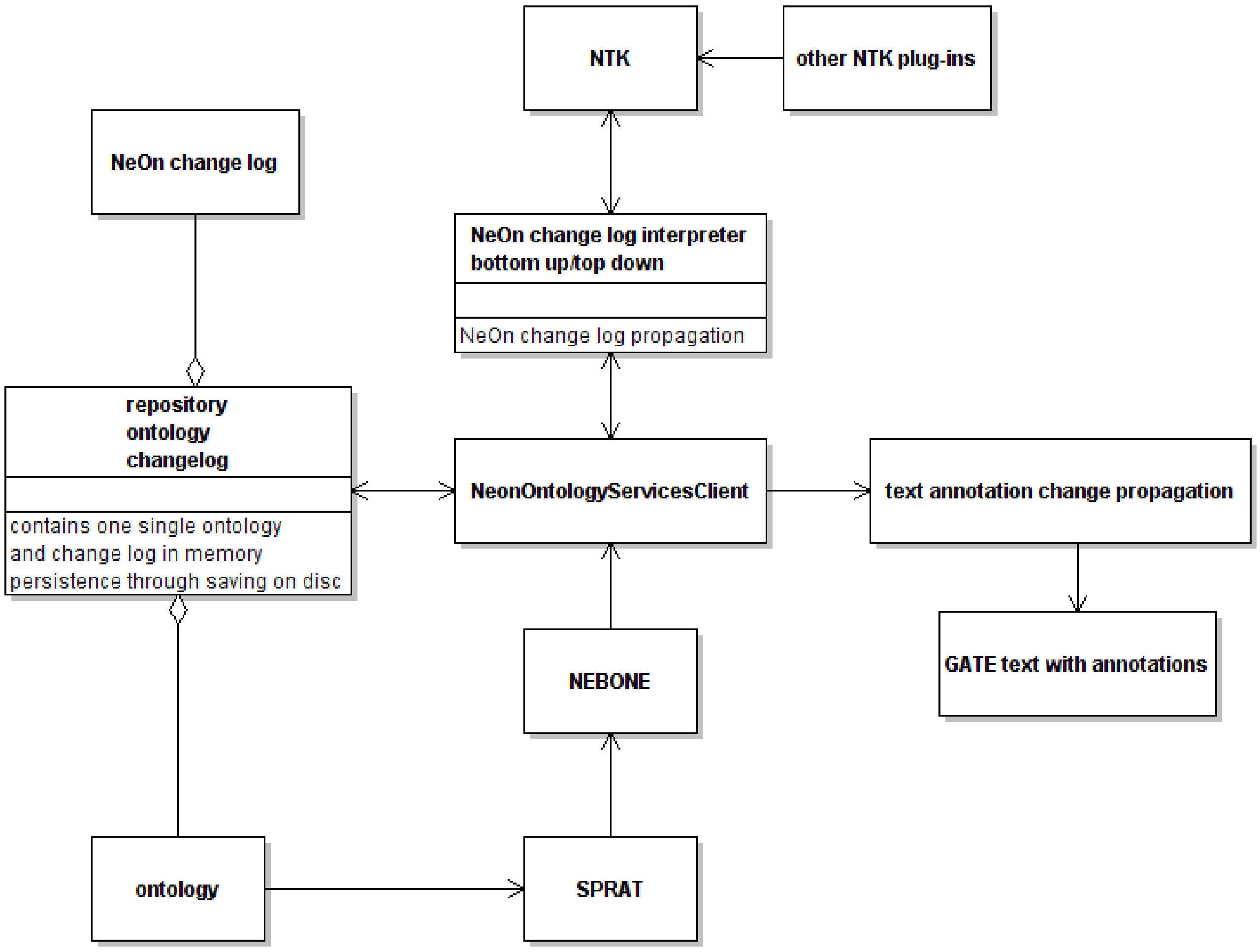

NTK
other NTK plug-ins
NeOn change log
NeOn change log interpreter bottom up/top down
NeOn change log propagation
repository ontology changelog
contains one single ontology and change log in memory persistence through saving on disc
NeonOntologyServicesClient
text annotation change propagation
GATE text with annotations
NEBONE
ontology
SPRAT

Figure 2.2: Interaction between GATE, the change logs and the NeOn toolkit

The system works in both directions. Given this setup, users can choose to work on a system of their choice (NeOn Toolkit or GOS) and produce a change log that can be interpreted by both the toolkit and GOS to bring the ontologies in the same state.

As described in the deliverable D1.5.2, the *NeonOntologyServiceClient* uses the graphical interface of Gate Ontology Editor to allow users to create and populate new and existing ontologies. The same Ontology editor is also used by a standalone resource known as *OWLIMOntologyLR* [3] which is a part of the core GATE system. For testing purposes only, the new development has been carried out within the core GATE environment. Both the *NeonOntologyServiceClient* and the *OWLIMOntologyLR* are implementations of the same interface, the only difference between the two being that the former requires access to a server running GOS where it can create, store and edit ontologies, whereas the latter creates repositories in memory which are destroyed as soon as the LR is closed. Once the new changes in *OWLIMOntologyLR* have been tested, the next step will be to integrate these changes within the *NeONOntologyServiceClient*.

Changes in Change logs The basic difference between the two change logs (the previous version of GOS vs the current version) is the type of information included in the change logs. The earlier change logs focused on providing information on the affected triples (subject, predicate, objects). For example, if an instance was deleted in the earlier version, the following three entries were added into the change log:

```
<instanceURI> <*> <*>
<*> <instanceURI> <*>
<*> <*> <instanceURI>
```

What it meant was that all the statements where <instanceUri> appeared as either subject, predicate or object were deleted. However, it lacked the information about what the other two elements were apart from the <instanceURI> in those deleted triples. Although such an arrangement helped in keeping the change logs simple and easy to read, it omitted to mention what statements (with full details on subject, predicate and object) were present before the deletion of this particular instance.

In a scenario where different people are contributing their efforts into the final outcome, it is very important to know what was contributed by whom. The earlier version of GOS did not record any information about the authors responsible for changes. The newer version adds information about the authors along with the timestamps to record when the changes occurred.

In the new version, change logs are produced as valid RDF/XML documents where every change made to an ontology is an instance of some appropriate class.

2.2 Recording and interpreting changes

Below, we first give some details about the change ontology (OWLChanges.owl) that was developed in the previous deliverables D1.3.1 and D1.3.2, and explain various parts of it. We then provide more information on how changes are recorded by explaining a step-by-step process for some example changes. Finally, we explain the process of interpreting it back.

The change ontology, as explained earlier, has several concepts that define every possible change in the ontology. For example, whenever a new class is added, an instance of *AddClass* is created and added to the change log. This instance has several properties such as when the change occurred, who made the change, which ontology this change belongs to, the URI(s) of the affected resource(s) and so on.

Every change in the ontology can be classified as either an *Addition* or a *Removal*. An instance of a concept called *ChangeSpecification* is created for every Addition or Removal made to the ontology. This instance is associated with the axiom (such as the one explained above) that provides more details about the change itself. Since the changes are recorded in the ontology, it is difficult to say which change occurred first. However, in order to apply the changes, it is very important to know the order in which changes occurred. To solve this problem, every change is associated with the change that took place just before the latest change, using a property called *hasPreviousChange*. Iterating over the values of these properties helps in identifying the correct order in which these changes occurred.

[3] http://gate.ac.uk/sale/tao/splitch10.html#x12-34800010.3

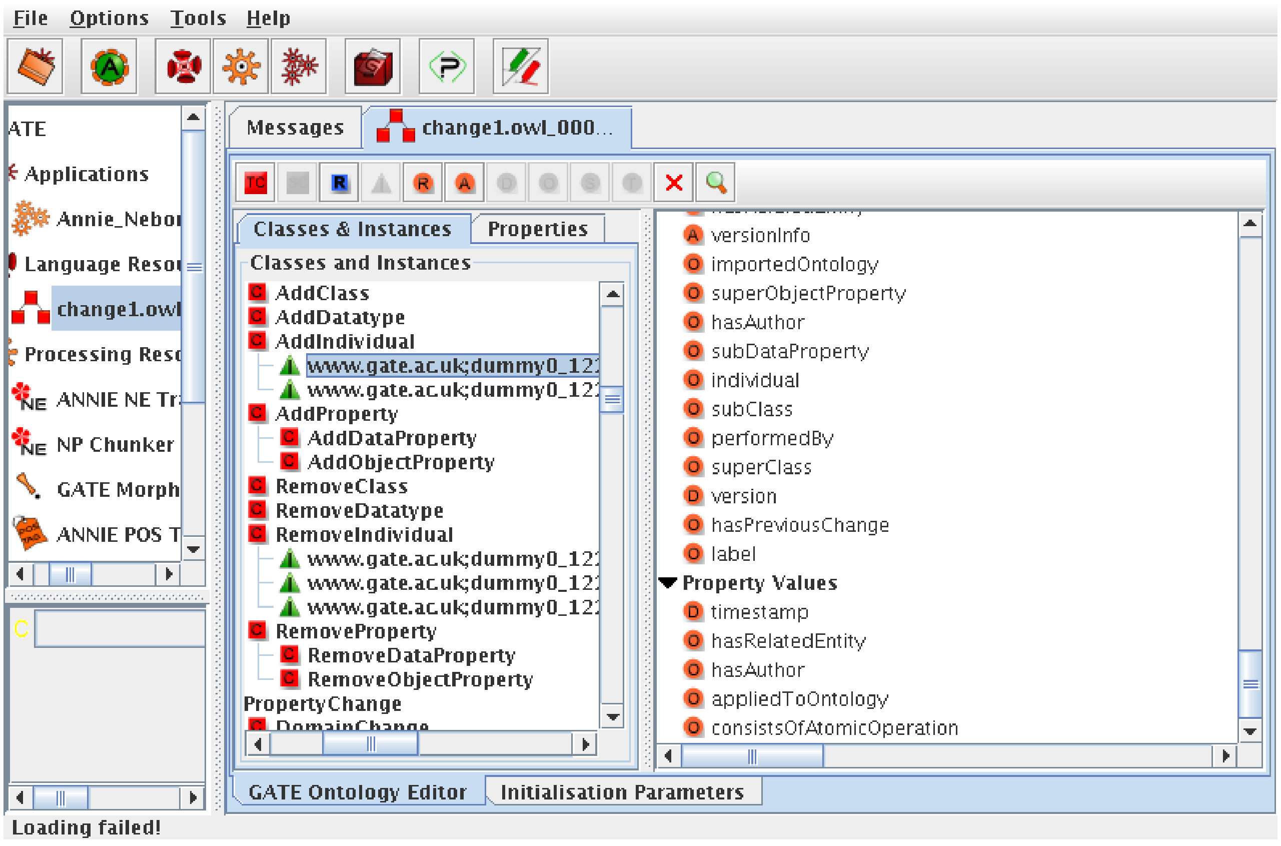


Figure 2.3: Change log saved after modifications to the initial ontology

In order to illustrate this process, let us consider an example. For example, a user wants to create a concept called *Person* with an instance *person1*. Below we describe the statements that are added at every step taken to register the change.

1. Addition of a concept *Person*

   (a) A random atomicChange uri is created with the following format:

   ```
   atomicChange = ontologyURL?location=ontologyBaseLocation;
   ontologyName;change=randomUniqueNumber
   ```

   (b) Based on the type of a change, the axiom is registered as an instance of either Addition or Removal:

   ```
   <atomicChange> <rdf:type> <Addition>
   ```

   (c) Information about when this atomic change took place:

   ```
   <atomicChange> <timestamp> <currentDateWithCurrentTime>
   ```

   (d) Information about which ontology this change belongs to:

   ```
   <atomicChange> <appliedToOntology> <ontologyURL>
   ```

   (e) Information about the author who made this change:

   ```
   <atomicChange> <hasAuthor> <userName>
   ```

(f) The subject *changeSpecification* in the following statement acts as a register of changes. A user wishing to read these changes should refer to this statement to obtain all the changes that took place in the ontology:

```
<ontologyUrl;changeSpecification> <hasChange> <atomicChange>
```

(g) An axiom is defined that gives information about the related/affected entities:

```
axiomDeclaration = ontologyURL?location=ontologyBaseLocation;
ontologyName;axiom=randomUniqueNumber
<axiomDeclaration> <rdf:type> <Declaration>
```

(h) The following triple tells the system about some related entity:

```
<axiomDeclaration> <entity> <Person>
```

(i) Relation between the axiom and atomic change is defined with the following statement:

```
<atomicChange> <appliedAxiom> <axiomDeclaration>
```

(j) Creating an instance of specific type of concept that explains this atomic operation:

```
atomicOperation = ontologyURL?location=ontologyBaseLocation;
ontologyName;change=randomUniqueNumber
<atomicOperation> <rdf:type> <AddClass>
```

(k) Which ontology this atomic operation is applied to:

```
<atomicOperation> <appliedToOntology> <ontologyURL>
```

(l) Which entity this atomic operation refers to:

```
<atomicOperation> <hasRelatedEntity> <Person>
```

(m) Who is the author of this atomic operation:

```
<atomicOperation> <hasAuthor> <userName>
```

(n) Finally, when exactly this atomic operation was registered:

```
<atomicOperation> <timestamp> <currentDateWithCurrentTime>
```

2. *person1* is an instance of *Person*. This operation consists of two atomic changes. The first change specifies that *person1* is an instance. The second change specifies that the *person1* is an instance of the concept *Person*.

(a) Repeat steps 1.1 to 1.7.

(b) In order to apply the change log it is necessary to know which operation took place first. The following statement helps in identifying the correct order:

```
<atomicChange> <hasPreviousChange> <previousAtomicChange>
```

(c) The following triple tells the system about some related entity:

```
<axiomDeclaration> <entity> <person1>
```

(d) Relation between the axiom and atomic change is defined with the following statement:

```
<atomicChange> <appliedAxiom> <axiomDeclaration>
```

(e) Creating an instance of specific type of concept that explains this atomic operation:

```
atomicOperation = ontologyURL?location=ontologyBaseLocation;
ontologyName;change=randomUniqueNumber
<atomicOperation> <rdf:type> <AddIndividual>
```

(f) Repeat steps 1.11 to 1.14.

(g) Repeat steps 1 and 2.

(h) An axiom is defined that gives information about the related/affected entities:

```
<axiomDeclaration> <rdf:type> <ClassAssertion>
```

(i) What are the individual and superclass names:

```
<axiomDeclaration> <individual> <person1>
<axiomDeclaration> <class> <Person>
<atomicChange> <appliedAxiom> <axiomDeclaration>
```

Because every change log is an ontology, it is possible to store each one under a separate ontology repository in GOS. As shown above, different statements are added to this repository for registering different changes in the ontology. As explained in D1.5.2, GOS allows the exporting of ontologies in different formats. Thus the change log can be exported in formats such as RDF/XML, NTriples, N3 and Turtle. By default, the change logs are exported as RDF/XML. Having exported change logs in one of the above formats, one can easily load them in any ontology editor to see the change log as a separate ontology. Figure 2.3 depicts a snapshot of the change log in the GATE Ontology Editor. It shows details such as the timestamp, username, affected resource etc. of the first change made to the ontology.

Having done this, the next task is to load such change logs back and apply them over to ontologies. GOS has an option that allows users to perform this task. Below, we show how these changes are read from the changelog..

Since the instances of the concept *ChangeSpecification* are used for recording the order in which these changes took place, the first step is to obtain all the instances of this concept and sort them to find out which change occurred first. As shown below, the first two triples are queried to obtain values of the *?change* variable. These are the values which provide pointers to the axioms and then to the atomicOperations (steps 4 and 5) with more information on changes. The third triple is used for finding out the order in which these changes occurred. Based on the type of operations (e.g. AddClass or AddIndividual) relevant information is obtained using the *?operationType*.

```
?a <rdf:type> <ChangeSpecification>.
?a <hasChange> ?change.
?change <hasPreviousChange> ?previousChange.
?change <consistsOfAtomicOperation> ?atomicOperation.
?atomicOperation <rdf:type> ?operationType.
```

Figure 2.4 shows part of an ontology created automatically by the GATE application ANNIE (which performs named entity recognition and translates these annotations into ontology elements) before the change log was applied. It finds in text instances of the concepts Person, Organization, Location etc. and adds these to an ontology. For example, we can see that it has found the names of people such as "Andrew Lloyd Webber" and "Doctor Who". It has also erroneously identified some names such as "Peggy Sue What" and "F BBC". After the ontology has been modified manually, removing the erroneous instances and adding some new ones (such as the "John Smith" instance mentioned earlier), we save the change log and then apply it to the previous version of the ontology. Figure 2.5 depicts a portion of the initial ontology after the change log was applied. We can clearly see the additions, deletions and changes made to the instances of Person.

2.3 Compatibility Testing

The development of the GOS change logs presented in this chapter is very new and needs a thorough compatibility testing with both the GOS and the NeOn Toolkit systems. At least four tests need to be carried out

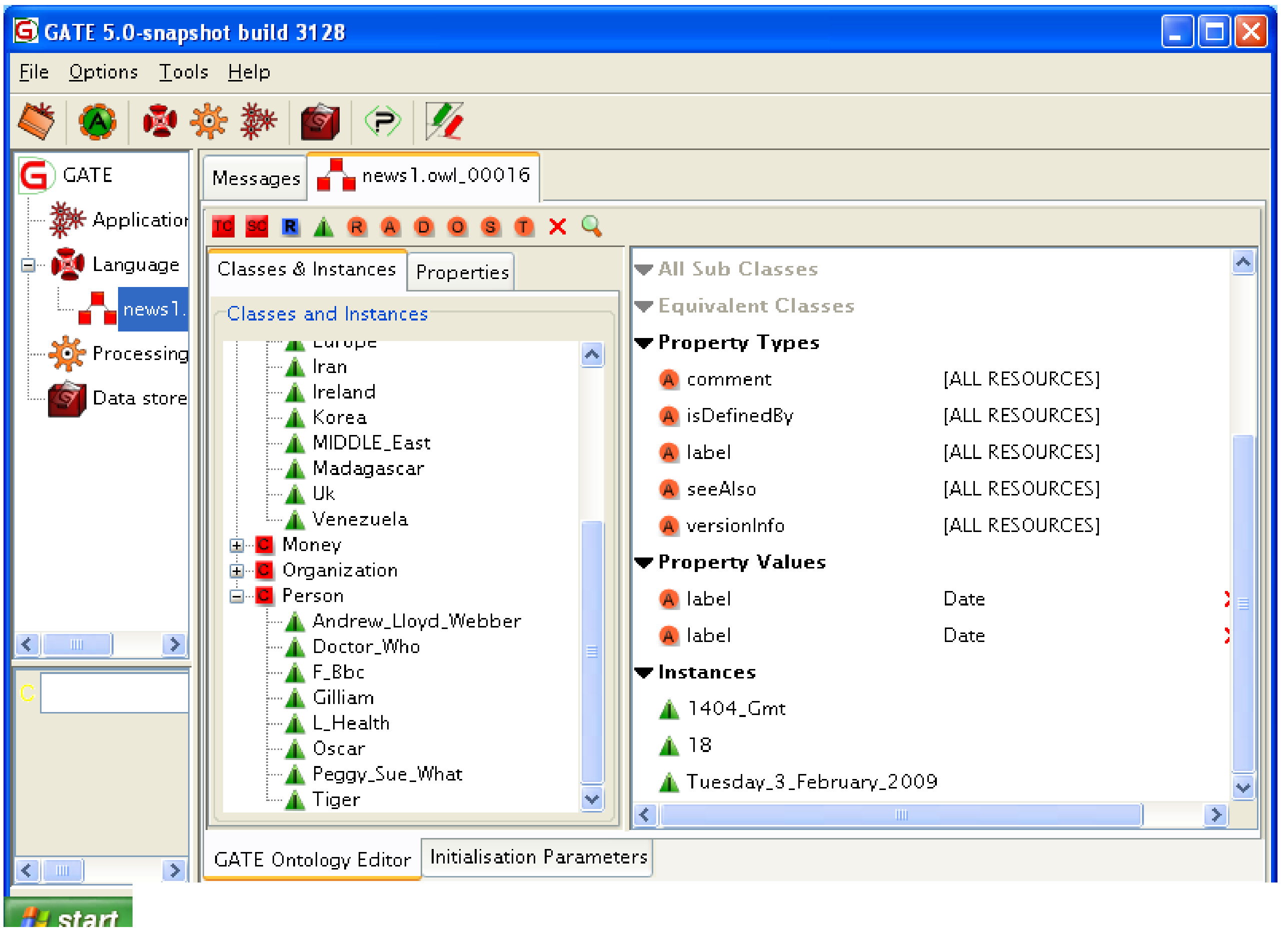


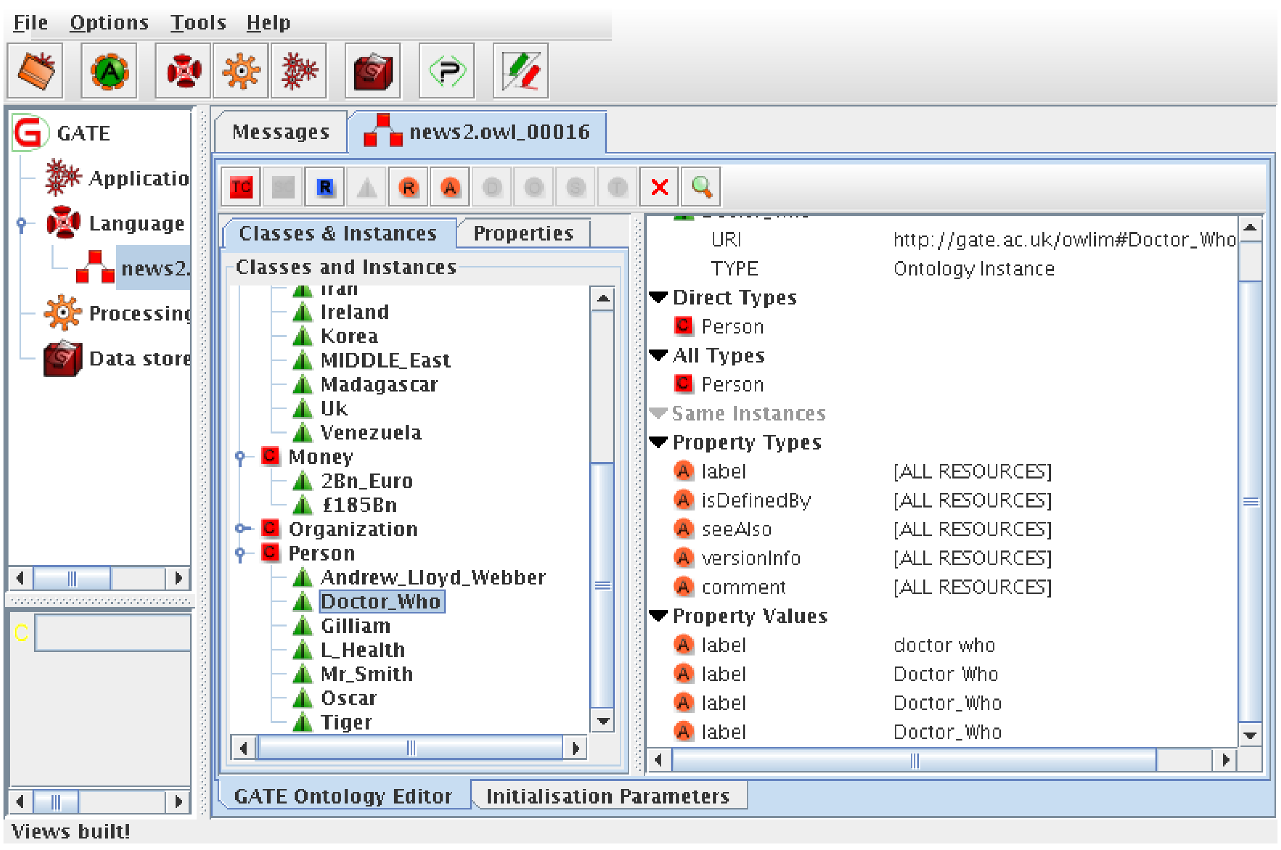
Figure 2.4: Ontology in GATE before change log application

to test the system. One of these involves producing a change log from GOS and checking if it is a valid RDF/XML. Since all the statements are added to a separate ontology repository, adding an invalid statement would cause immediate complaint. Since they do not produce complaint, we know that the ontologies (change logs) are guaranteed to be valid RDF/XMLs. The second test is to produce a change log from GOS and apply it back in GOS. The test would be successful if the changes registered in the change log are applied successfully in the correct order. Our system has successfully passed this test. The third test is to take a change log produced from the NeOn Toolkit and apply it over to the ontology in GOS. Finally, the fourth test is to take a change log produced from GOS and apply it over to the ontology in NeOn Toolkit. The last two changes are yet to be carried out and will form part of the work planned for the coming months.

2.4 Download

As described earlier, the decision was made to make these changes available from the *OWLIMOntologyLR* for now. This is because not only is it easier to test new changes with the *OWLIMOntologyLR*, which does not require setting up the GOS server, but also it makes it simpler for other people to try the new changes. Therefore the current download does not include an implementation of the *NeonOntologyServiceClient*. Once the changes in *OWLIMOntologyLR* have been fully tested, the new changes will be integrated in the *NeonOntologyServiceClient*, which will be the final outcome of this work.

In order to download and test the new changes, the following steps shold be taken:



URI	http://gate.ac.uk/owlim#Doctor_Who
TYPE	Ontology Instance
Direct Types	
Person	
All Types	
Person	
Same Instances	
Property Types	
label	[ALL RESOURCES]
isDefinedBy	[ALL RESOURCES]
seeAlso	[ALL RESOURCES]
versionInfo	[ALL RESOURCES]
comment	[ALL RESOURCES]
Property Values	
label	doctor who
label	Doctor Who
label	Doctor_Who
label	Doctor_Who

Figure 2.5: Ontology in GATE after change log application

Get the sources

1. Download a copy of GATE from http://gate.ac.uk/download/index.html.
2. Download the OWLIM zip file from http://www.gate.ac.uk/projects/neon/changelog.html
3. Unzip this file to replace the Java files in src/gate/creole/ontology folder.
4. Rebuild gate by calling "bin/ant all".

Note that the software needs to be built by the user before running because it makes some changes to the default GATE code. For reasons of backwards compatibility and possible influences on other GATE behaviour, it is better to keep these changes separate rather than rolling them into the default GATE code. It is standard behaviour that external GATE plugins are downloaded and built in this way.

Generating and Exporting Change logs

1. Start Gate by callling “bin/ant run”.
2. Load the "Ontology Tools" plugin.
3. Right click on the "Language Resources" and select the "OWLIM Ontology LR". This will bring up a window that will allow you to load an ontology. More information on how to use this Language Resource is available at http://gate.ac.uk/sale/tao/splitch10.html#x12-34800010.3.

4. As you make changes in the ontology, the changes are recorded in the background. For this prototype only, the changes are recorded in memory. You can download these changes any time by right clicking on the instance of ontology under "Language Resources" and selecting the "Save Ontology Event Log" option. When asked, select a location and filename to save the changes to a file. This file is an ontology itself which can be loaded in the same way that other ontologies are loaded in GATE.

5. In order to test the application of change logs, close GATE and repeat the steps from 1 to 3. This will load the original ontology. In order to apply the previously saved change log, right click on the instance of ontology under "Language Resources" and select the "Load Ontology Event Log" option. When asked, select the change log and click on the "Open" button. The changes in the change log will be applied and the editor will refresh it to show the modified ontology.

2.5 Licencing and Technical Support

Since the software forms part of the GATE architecture, it is supported by the regular GATE support infrastructure. This is detailed more explicitly on the GATE website at http://gate.ac.uk/support.html. Primarily, the GATE team should be contacted via the links on the website, or via the GATE users mailing list[4]. There is also detailed documentation provided on the main GATE web pages[5] and in the User Guide[6]. With respect to licencing, GATE is available under the LGPL licence and is available to use for both commercial and research purposes.

[4]http://gate.ac.uk/mail/index.html
[5]http://gate.ac.uk
[6]http://gate.ac.uk/sale/tao

Chapter 3

Evolva: Ontology Evolution Plugin for the NeOn Toolkit

3.1 Introduction

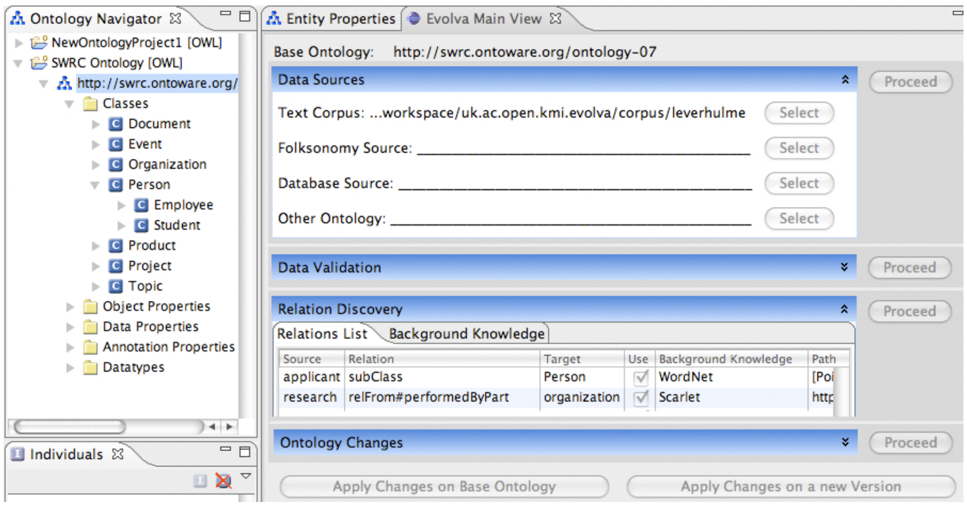
Ontology evolution is increasingly getting research momentum in the Semantic Web field. This is due to the fact that ontologies, forming the backbone of Semantic Web systems, need to be kept up-to-date so that ontology-based systems remain usable. We highlight two research approaches in the domain. The first considers ontology evolution as a pure management of changes performed by the user [Kle04, NCLM06, Sto04, VPST05], while the second takes into account dynamically updating and learning ontologies without handling the management of changes and ontology consistency [AHO06, BHSV06, NLH07]. We understand ontology evolution as the "timely adaptation of an ontology to the arisen changes and the consistent management of these changes" [HS05]. This definition indirectly reflects the need of combining the two aforementioned approaches for achieving a successful evolution. Yet no practical and complete solutions exist that cover all stages of evolution.

We describe in this chapter our ontology evolution framework with a pilot system implemented as a NeOn toolkit plugin. We focus on the evolution of ontologies from external data sources (e.g. text documents and folksonomy), through mainly relying on background knowledge sources to lift user input. While in deliverable 1.6.1, machine learning techniques are used to predict ontology structure changes from a corpus of documents, here we rely on knowledge reuse from external background knowledge sources for linking new knowledge to the ontology. Our proposed framework, described in the next section, is designed to cover a complete ontology evolution cycle. However, being aware of the amount of research involved in each step, some of the features are planned to be reused from existing work (e.g. change recording and propagation in D1.3.1).

3.2 Approach and Implementation

We are planning to close the above gap by proposing Evolva, a complete ontology evolution framework that covers the entire evolution cycle, and makes use of background knowledge to potentially decrease, or even eliminate, user involvement. The need for Evolva emerged from the tedious and time consuming update and evolution of our KMi Semantic Web portal[1] ontology. Being highly user dependent and occurring in a dynamic domain, the ontology was left outdated. Figure 3.1 illustrates a screenshot of Evolva's pilot system.

[1] http://semanticweb.kmi.open.ac.uk/



Base Ontology: http://swrc.ontoware.org/ontology-07

Text Corpus: ...workspace/uk.ac.open.kmi.evolva/corpus/leverhulme

Folksonomy Source: ______

Database Source: ______

Other Ontology: ______

Source	Relation	Target	Use	Background Knowledge	Path
applicant	subClass	Person	✓	WordNet	[Poi
research	relFrom#performedByPart	organization	✓	Scarlet	http

Figure 3.1: Evolva Pilot System Screenshot

3.2.1 Ontology Evolution Framework

Evolva is a complete ontology evolution framework that relies on various sources of background knowledge to support its process. Below we provide a brief overview of its five components design, depicted also in Figure 3.2, which is partially implemented. More details are available in [Zab08].

1. **Information Discovery**. Our approach starts with discovering potentially new information from the data sources associated with the information system. Contrasting the ontology with the content of these sources is a way of detecting new knowledge that should be reflected by the base ontology. Data sources exist in various formats from unstructured data such as text documents or tags, to structured data such as databases and ontologies. This component handles each data source differently. (1) Text documents are processed using information extraction, ontology learning or entity recognition techniques. (2) Other external ontologies are subject to translation for language compatibility with the base ontology, and (3) database content is translated into ontology languages.

2. **Data Validation**. Discovered information is validated in this component. We rely on a set of heuristic rules such as the length of the extracted terms. This is especially needed for information discovered from text documents, as information extraction techniques are likely to introduce noise. For example, most of the two-letter terms extracted from KMi's news corpora are meaningless and should be discarded. In the case of structured data, this validation is less necessary as the type of information is explicitly defined.

3. **Ontological Changes**. This component is in charge of establishing relations between the extracted terms and the concepts in the base ontology. These relations are identified by exploring a variety of background knowledge sources, as we will describe in the next section. Appropriate changes will be represented and performed to the base ontology, generating a new ontology version.

4. **Evolution Validation**. Performing ontology changes automatically may introduce inconsistencies and incoherences in the base ontology. Also, due to having multiple data sources, data duplication is likely to arise. Conflicting knowledge is highly possible to occur and should be handled by automated reasoning. As evolution is an ongoing process, many statements are time dependent and should be treated accordingly by applying temporal reasoning techniques.

5. **Evolution Management**. Managing the evolution will be about giving the ontology curator a degree of control over the evolution, as well as propagating changes to the dependent components of the

ontology such as other ontologies or applications. User control will deal with tracking ontology changes, spotting and solving unresolved problems.

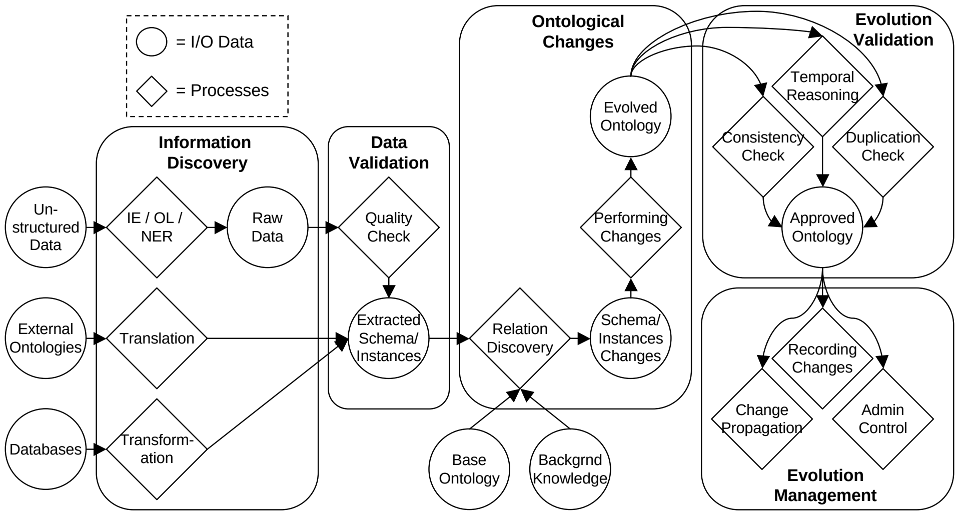


Figure 3.2: The Main Components of Evolva

3.2.2 The Role of Background Knowledge in Evolva

A core task in most ontology evolution scenarios is the integration of new knowledge into the base ontology. We focus on those scenarios in which such new knowledge is extracted as a set of emerging terms from textual corpora, databases, or domain ontologies. Traditionally this process of integrating a new set of emerging terms is performed by the ontology curator. For a given term, they would rely on their own knowledge of the domain to identify in the base ontology elements related to the term, as well as the actual relations they share. As such, it is a time consuming process, which requires the ontology curator to know well the ontology, as well as being an expert in the domain it covers.

Evolva makes use of various background knowledge sources to identify relations between new terms and ontology elements. The hypothesis is that a large part of the process of updating an ontology with new terms can be automated by using these sources as an alternative to the curator's domain knowledge.

We have identified several potential sources of background knowledge. For example, thesauri such as WordNet [Fel98] have been long used as a reference resource for establishing relations between two given concepts, based on the relation that exists between their synsets. Because WordNet's dictionary can be downloaded and accessed locally by the system and because a variety of relation discovery techniques have been proposed and optimized, exploring this resource is quite fast. Online ontologies constitute another source of background knowledge which has been recently explored to support various tasks such as ontology matching [SdM08] or enrichment [ASSM07]. While the initial results in employing these ontologies are encouraging, these techniques are still novel and in need of further optimizations (in particular regarding time-performance). Finally, the Web itself has been recognised as a vast source of information that can be exploited for relation discovery through the use of so-called lexico-syntactic patterns [CHS04]. Because they rely on unstructured textual sources, these techniques are more likely to introduce noise than the previously mentioned techniques which rely on already formalized knowledge. Additionally, these techniques can be time consuming given that they operate at Web scale.

Taking into account these considerations, we have devised a relation discovery process that combines various background knowledge sources with the goal of optimising time-performance and precision. As shown in Figure 3.3, the relation discovery starts from quick methods that are likely to return good results, and continues with slower methods which are likely to introduce a higher percentage of noise, via the following steps.

1. The process begins with string matching for detecting already existing terms in the ontology. This will identify equivalence relations between the new terms and the ontology elements.
2. Extracted elements that do not exist in the base ontology are passed to a module that performs relation discovery by exploring WordNet's synset hierarchy.
3. Terms that could not be incorporated by using WordNet are passed to the next module which explores Semantic Web ontologies.
4. If no relation is found, we resort to the slower and more noisy methods which explore the Web itself through search engine APIs and lexico-syntactic patterns [CHS04]. In case no relation is found at the final level, the extracted term is discarded or, optionally, forwarded for manual check.

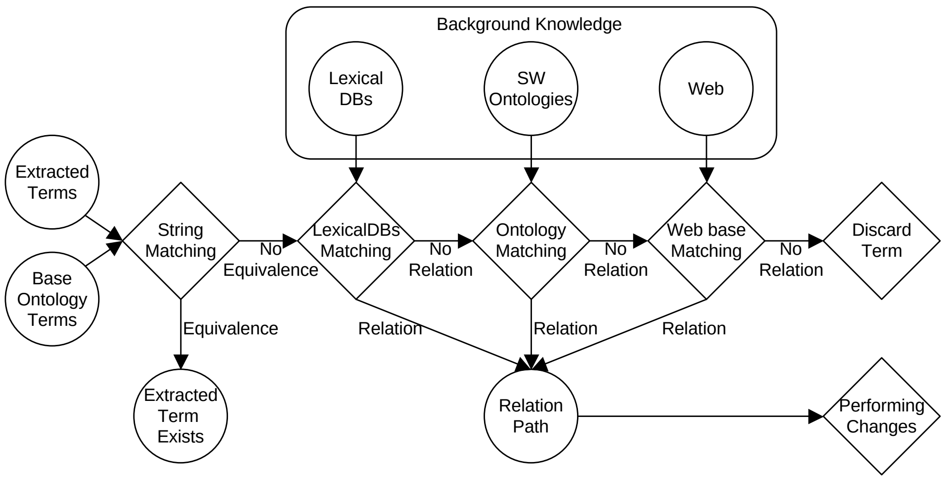


Figure 3.3: Finding relations between new terms and the base ontology in Evolva.

3.2.3 Implementation of Evolva's Relation Discovery

We have partially implemented the algorithm presented in Figure 3.3 by making use of methods for exploring two main background knowledge sources: WordNet and online ontologies. We have not yet implemented methods for exploiting the Web as a source of knowledge. The first part of the implementation performs the string matching between the extracted terms and the ontology elements. We rely on the Jaro distance metric similarity [CRF03] which takes into account the number and positions of the common characters between a term and an ontology concept label. This string similarity technique performs well on short strings, and offers a way to find a match between strings that are slightly different only because of typographical errors or the use of different naming conventions.

For the WordNet based relation discovery, we derive a relation by exploring WordNet's hierarchy using a functionality built into its Java library[2]. This will result in a relation between a term, as well as an inference path which lead to its discovery.

The terms that could not be related to the base ontology are forwarded to the next module which makes use of online ontologies. For this component, we rely on the Scarlet relation discovery engine[3]. It is worth noting that we handle ontologies at the level of statements, i.e. ontologies are not processed as one block of statements. Thus we focus on knowledge reuse without taking care of the validation of the sources as a whole with

[2]http://jwordnet.sourceforge.net/
[3]http://scarlet.open.ac.uk/

respect to the base ontology. Scarlet [SdM08] automatically selects and explores online ontologies to discover relations between two given concepts. For example, when relating two concepts labelled *Researcher* and *AcademicStaff*, Scarlet identifies (at run-time) online ontologies that can provide information about how these two concepts inter-relate and then combines this information to infer their relation. [SdM08] describes two increasingly sophisticated strategies to identify and exploit online ontologies for relation discovery. We rely on the first strategy which derives a relation between two concepts if this relation is defined within a single online ontology, e.g. stating that *Researcher* $\sqsubseteq$ *AcademicStaff*. Besides subsumption relations, Scarlet is also able to identify disjoint and named relations. All relations are obtained by using derivation rules which explore not only direct relations but also relations deduced by applying subsumption reasoning within a given ontology. For example, when matching two concepts labelled *Drinking Water* and *tap_water*, appropriate anchor terms are discovered in the TAP ontology[4] and the following subsumption chain in the external ontology is used to deduce a subsumption relation: *DrinkingWater* $\sqsubseteq$ *FlatDrinkingWater* $\sqsubseteq$ *TapWater*. Note, that as in the case of WordNet, the derived relations are accompanied by a path of inferences that lead to them.

We performed an experiment about the potential use of such background knowledge sources for relation discovery, and they proved to have a high precision of around 77%. Further details of the experiment can be found in [ZSdM08].

3.2.4 Implementation of the NeOn toolkit Plugin

The Evolva NeOn toolkit plugin is a concrete implementation of our framework (see screenshot in Figure 3.1). Currently, the plugin supports the first three steps depicted in Figure 3.2.

In the *Information Discovery* step, Text2Onto is used to identify concepts in a corpus of text documents (selected in the "Data Sources" panel). *Data validation* relies on string similarity and term length measures. We use a customizable Jaro-based [CRF03] calculation to compute the similarity of the extracted and existing concepts. Concepts with a similarity under a given threshold are considered new and are validated based on their length. The user can set the parameters of the validation techniques and also manually indicate which concepts should be considered for integration ("Data Validation").

Ontological changes are identified by finding links between validated terms and ontology concepts. We use two main sources of background knowledge: WordNet and online ontologies accessed through Scarlet[5]. As with WordNet, the derived relations are accompanied by a path of inferences that lead to them ("Relation Discovery" panel). Figure 3.1 shows an example of how WordNet helped linking the new concept *Applicant* as a *subClassOf Person* (a concept in the base ontology). A second example shows how Scarlet links *Website* to *Organization*, through a *hasWebsite* relation. One of the challenges at this level is to validate the relations efficiently, prior to applying any changes on the base ontology. For example, we need to know how to select the right synset in WordNet, or how to determine whether a relation discovered from online ontologies conflicts with the existing knowledge in the base ontology. Currently we are relying on the web-based distance similarity measure [CV07] as a step to check the possibility of two terms being related, before performing relation discovery. Part of our future plan is to use other validation techniques such as the base ontology itself as a validator, as well as word sense disambiguation. In addition to these automated validation methods, the user can manually exclude irrelevant relations. The user is allowed to manually select the relations to be used. A list of ontology changes is deduced from these relations and applied to the ontology. Changes are represented and performed using the Change Ontology developed within the NeOn toolkit and discussed in Section 2.1. For further details of the ontology, see D 1.3.1.

3.3 Download and Documentation

The current pilot Evolva plugin is still under major development and testing. Tests have currently been performed only on MacOS, and we are in the process of testing the plugin in other environments. The pilot plugin

[4]http://139.91.183.30:9090/RDF/VRP/Examples/tap.rdf
[5]http://scarlet.open.ac.uk/

is available for download, with online installation procedures and user documentation which includes how to start Evolva and proceed throughout the ontology evolution process. The download and documentation are available on the Evolva website[6].

3.4 Future Work

Part of our future plan is to study automated techniques for relation validation, crucial to increase Evolva's precision. We also plan to fine-tune and extend our prototype to cover the remaining framework components. This includes (1) evolution validation for consistency and duplication checks that could have occurred as an evolution side effect, and (2) the evolution management for recording changes and handling change propagation to the dependent components such as applications, or imported and aligned ontologies. Recording changes will be based on the Change Capturing plugin of the NeOn toolkit (see D1.3.1).

We plan to test Evolva in real domains such as the UN's Food and Agriculture Organization (FAO)[7], linking with work performed in WP7. This would give us firstly qualitative measures, which reflect the correctness of the content added to the ontology; and secondly, quantitative measures to analyze the performance of Evolva in terms of time, number of ontology entities added, and number of sources analyzed.

[6]http://evolva.kmi.open.ac.uk
[7]http://www.fao.org

Chapter 4

Conclusions and Future Work

In this deliverable we have described the implementation of the approach to modelling some of the dynamics of (semantic) metadata that were described in previous NeOn deliverables. On the one hand, we have achieved interoperability between the NeOn toolkit and the GATE architecture where ontology changes are concerned, so that changes to an ontology made by different users can be merged and/or made available to others. The reason that this is necessary is because although GATE services such as SPRAT, SARDINE and ANNIE can be run as NeOn toolkit plugins, these are only loosely coupled.

On the other hand, we have implemented another plugin, Evolva, which demonstrates the bottom-up approach to ontology change, through the use of evolving metadata from text and folksonomies. In future work, we hope to incorporate into Evolva some of the approaches proposed in the previous deliverable D1.5.2, such as that used in the SPRAT application (described more fully in [VTSFGP+08]).

One thing which is still missing in this work is a proper integration of the change management process and the GATE web service applications such as SPRAT and SARDINE. When for example SPRAT is run on a set of documents, and a previously existing ontology is modified, the change log is not automatically saved but must be manually created. In future, this will be incorporated into the application when appropriate, so that the changelog is saved as well as the new version of the ontology, and thus when loaded in the toolkit, a previous version of the original ontology (stored on another server by another user, for instance) can be modified with the new version. This is important for collaborative ontology development work.

Bibliography

[AHO06] H. Alani, S. Harris, and B. O'Neil. Winnowing ontologies based on application use. In *Proceedings of ESWC*, 2006.

[ASSM07] S. Angeletou, M. Sabou, L. Specia, and E. Motta. Bridging the gap between folksonomies and the semantic web: An experience report. In *Proc. of the ESWC Workshop on Bridging the Gap between Semantic Web and Web 2.0*, 2007.

[BHSV06] Stephan Bloehdorn, Peter Haase, York Sure, and Johanna Voelker. Ontology evolution. In *Semantic Web Technologies - Trends and Research in Ontology-based Systems*, pages 51–70. John Wiley & Sons, June 2006.

[CHS04] P. Cimiano, S. Handschuh, and S. Staab. Towards the self-annotating web. In *Proceedings of the 13th international conference on World Wide Web*, pages 462–471, 2004.

[CRF03] W. W. Cohen, P. Ravikumar, and S. E. Fienberg. A comparison of string distance metrics for name-matching tasks. In *Proceedings of the IJCAI-2003 Workshop on Information Integration on the Web (IIWeb-03)*, 2003.

[CV07] R. L. Cilibrasi and P. M. B. Vitányi. The google similarity distance. In *IEEE Transactions on Knowledge and Data Engineering*, pages 370–383, 2007.

[Fel98] Christiane Fellbaum, editor. *WordNet - An Electronic Lexical Database*. MIT Press, 1998.

[HS05] P. Haase and L. Stojanovic. Consistent evolution of owl ontologies. In *Proceedings of ESWC*, pages 182–197, 2005.

[Kle04] M. Klein. *Change Management for Distributed Ontologies*. PhD thesis, Vrije Universiteit in Amsterdam, 2004.

[MPAD08] D. Maynard, W. Peters, S. Angeletou, and M. D'Aquin. Implementation of metadata evolution. Technical Report D1.5.2, NeOn Project Deliverable, 2008.

[MPD$^+$07] D. Maynard, W. Peters, M. D'Aquin, M. Sabou, and N. Aswani. Dynamics of metadata. Technical Report D1.5.1, NeOn Project Deliverable, 2007.

[NCLM06] N. F. Noy, A. Chugh, W. Liu, and M. A. Musen. A framework for ontology evolution in collaborative environments. In *Proc. of ISWC*, pages 544–558, 2006.

[NLH07] V. Novacek, L. Laera, and S. Handschuh. Semi-automatic integration of learned ontologies into a collaborative framework. In *International Workshop on Ontology Dynamics*, 2007.

[PHJ09] R. Palma, P. Haase, and Q. Ji. Change management to support collaborative workflows. Technical Report D1.3.2, NeOn Project Deliverable, 2009.

[PHWD08] R. Palma, P. Haase, Y. Wang, and M. D'Aquin. Propagation models and strategies. Technical Report D1.3.1, NeOn Project Deliverable, 2008.

[SdM08] M. Sabou, M. d'Aquin, and E. Motta. Exploring the semantic web as background knowledge for ontology matching. *Journal on Data Semantics*, 2008.

[Sto04] L. Stojanovic. *Methods and Tools for Ontology Evolution*. PhD thesis, FZI, 2004.

[VPST05] D. Vrandecic, H. S. Pinto, Y. Sure, and C. Tempich. The diligent knowledge processes. *Journal of Knowledge Management*, 9:85–96, 2005.

[VTSFGP$^+$08] B. Villazón-Terrazas, M. Suárez-Figueroa, A. Gómez-Pérez, A. García-Silva, and D. Maynard. Methods and tools for re-engineering non-ontological resources. Technical Report D2.2.2, NeOn Project Deliverable, 2008.

[Zab08] F. Zablith. Dynamic ontology evolution. In *ISWC Doctoral Consortium*, 2008.

[ZSdM08] F. Zablith, M. Sabou, M. d'Aquin, and E. Motta. Using background knowledge for ontology evolution. In *International Workshop on Ontology Dynamics*, 2008.