A TOOL TO SUPPORT ONTOLOGY CREATION BASED ON INCREMENTAL MINI-ONTOLOGY MERGING

by

Zonghui Lian

A thesis submitted to the faculty of

Brigham Young University

in partial fulfillment of the requirements for the degree of

Master of Science

Department of Computer Science

Brigham Young University

March 2008

BRIGHAM YOUNG UNIVERSITY

GRADUATE COMMITTEE APPROVAL

of a thesis submitted by

Zonghui Lian

This thesis has been read by each member of the following graduate committee and by majority vote has been found to be satisfactory.

Date	David W. Embley, Chair
Date	Stephen W. Liddle
Date	Charles D. Knutson

ABSTRACT

A TOOL TO SUPPORT ONTOLOGY CREATION BASED ON INCREMENTAL MINI-ONTOLOGY MERGING

Zonghui Lian

Department of Computer Science

Master of Science

This thesis addresses the problem of tool support for semi-automatic ontology mapping and merging. Solving this problem contributes to ontology creation and evolution by relieving users from tedious and time-consuming work. This thesis shows that a tool can be built that will take a “mini-ontology” and a “growing ontology” as input and make it possible to produce manually, semi-automatically, or automatically an extended growing ontology as output. Characteristics of this tool include: (1) a graphical, interactive user interface with features that will allow users to map and merge ontologies, and (2) a framework supporting pluggable, semi-automatic, and automatic mapping and merging algorithms.

ACKNOWLEDGMENTS

First of all, I would like to thank my advisor, Dr. David W. Embley. Under his guidance, I successfully overcame many difficulties and learned a lot from him.

Secondly, I would like to thank my other committee members. I thank Dr. Stephen W. Liddle for his unique insight into software design and implementation. I thank Dr. Charles D. Knutson for his time and effort on my thesis.

I would like to thank my wife, Cui, for her unwavering support and encouragement, as well as her patience and understanding. I thank my son, Andrew, for bringing me so many happiness. I also want to thank my parents and my brother, Zongyang, for their support and help all the time in my life.

I thank the National Science Foundation for supporting this research under grant No.0414644

Last, but not least, I thank all the BYU data-extraction research group members for their support and suggestions on my research.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

Ontologies provide a powerful explanatory mechanism for concepts and their relationships. The creation of ontologies, however, is expensive and complex. In most cases, information collection and analysis, concept and relationship design, and iterative construction are tedious and time-consuming work.

Much effort has been expended to alleviate these difficulties. Currently over 90 ontology editor tools are available [8]. Although focused on different purposes and based on different ontology languages, we can generally classify these ontology development tools into three categories according to their dominant creation processes: manual, semi-automatic, and automatic.

Manual ontology generation tools, such as [1], [2], [3], [5], [13], and [17] provide a graphical user interface for users to create ontologies. These tools are relatively simple: they provide no information extraction and analysis functions, and most of them provide no support for ontology merging. Although these tools provide some conveniences, they still leave most of the information collection and analysis work to be done manually.

Some tools are semi-automatic, providing automatic support only for some parts of ontology creation [18]. For example, based on machine learning and linguistic processing, Text2Onto [7] can extract concepts and relations from textual data. A limitation of this tool is the difficulty of creating the corpus necessary to support machine learning and natural language processing techniques. If users want to use this tool to create an ontology in a certain domain, they must have enough data in the corpus first, which in most cases is hard to obtain. Also, the resulting ontologies are often not reliable, and in some cases, the accuracy of the relationships among extracted data items is low. Lamparter, et al. [15] present another example of a semi-automatic ontology creation method. This method allows semi-automatic knowledge extraction from underlying classification schemas such as folder structures or web directories. The performance of this method depends highly on previously generated ontologies. In addition, the accuracy of distinguishing concepts and data instances is a problem for this method.

There are also some available tools that provide for automatic ontology generation. However, they are all highly constrained. TOPKAT [4] supplies a simple natural language parser that only provides partial natural language analysis to identify possible concepts and property values. The tool presented by Modica, et al. [20] is able to extract ontologies only if the scope of the domain is very narrow. Although the authors claim that this tool is automatic, an ontology evolution session is highly user-interactive.

Researchers at BYU have proposed TANGO (Table ANalysis for Generating Ontologies), which can generate ontologies fully automatically, but can also be run semi-automatically or manually. TANGO [22] is an ontology creation tool that assembles information provided in ordinary tables into ontologies. The aim of TANGO is to create ontologies while involving the user as little as possible. TANGO's working process includes four steps: (1) recognize and canonicalize table information; (2) construct mini-ontologies[1] from canonicalized tables; (3) discover inter-ontology mappings; and (4) merge mini-ontologies into a growing application ontology[2].

My research is to construct a tool to do the ontology mapping and merging, which is a part of the TANGO project (steps 3 and 4). Specifically, the input for a session using this tool is a mini-ontology and a growing ontology, while the output is the growing ontology augmented by the mini-ontology. The tool should allow the ontology mapping and merging processes to be automated. It should also guide users by notifying them when interventions may be necessary and by suggesting possible

[1]A *mini-ontology* is a lightweight ontology generated from a table by table interpretation techniques.

[2]A *growing ontology* is a domain ontology that is being constructed as a result of merging mini-ontologies into a growing ontology.

resolutions to questions and ambiguities that may arise. Characteristics of this tool include: (1) a graphical, interactive user interface with features that will allow users to map and merge ontologies, and (2) a framework supporting pluggable, semi-automatic and automatic mapping algorithms.

This tool can help users to resolve ontology mapping and merging problems with its user-friendly interface, and it can be extended by plugging in different mapping and merging algorithms. With these algorithms, it even can resolve the mapping and merging issues semi-automatically or automatically. As a part of the TANGO project, this approach will ease the difficulties of creating ontologies.

There exist many ontology mapping tools that support graphical user interfaces [11, 12]. For example, COMA++ [9], OLA [14], Clio [19], and PROMPT [21] all provide ontology mappings semi-automatically. These tools mainly focus on the OWL ontology mapping and the schema mapping. Our tool focuses on both schema mapping and relationship set mapping. In addition, our tool aims to map and merge extraction ontologies that have more ontology components than do OWL ontologies. We give more details about extraction ontologies in Chapter 2.

We give the details of this tool as follows. Chapter 2 introduces basic knowledge about ontologies and the OntologyEditor tool. It also explains how to perform merging and mapping between two ontologies so they can be used within a single model. Chapter 3 demonstrates usage of the tool. Chapter 4 describes the API (application programming interface) for plug-in mapping and merging algorithms. Chapter 5 describes our experience with brief field tests and subsequent observations. Finally, Chapter 6 concludes the thesis and explores possibilities for future work.

Chapter 2

Preparatory Ontology Editor Augmentations

In this chapter, we briefly introduce ontologies and the OntologyEditor [23], the environment we use to generate and edit ontologies. In this thesis, we augment the OntologyEditor by adding merging and mapping functions. Originally the OntologyEditor was designed and developed for editing only one ontology at a time. In order to do ontology mapping and merging, however, we need to open and edit two ontologies at the same time. We therefore created an operation allowing a user to open two ontologies in the same view in preparation for mapping and merging.

2.1 Ontologies and the Ontology Editor

The structural components of an ontology include object sets, relationship sets, and constraints over these object and relationship sets. An object set in an ontology represents a set of objects which may either be lexical or nonlexical. A lexical object set contains object values. For example, the string "U.S.A." is a value of *Name* (Country Name), which is a lexical object set. A nonlexical object set describes an abstract concept, such as *Country.* A nonlexical object set contains object identifiers and usually has other associated object values to describe it. For example, we could use a country name to describe a country.

Figure 2.1 shows a graphical view of a sample ontology opened in the OntologyEditor [23], an ontology design and maintenance tool constructed by the Data Extraction Group at BYU. In the ontology, a dashed box represents a lexical object set (e.g., *Province*), and a solid box represents a nonlexical object set (e.g., *Country*). Lines connecting object sets represent relationship sets. A word or short phrase along with a reading-direction arrow and the connected object set names represents the re-

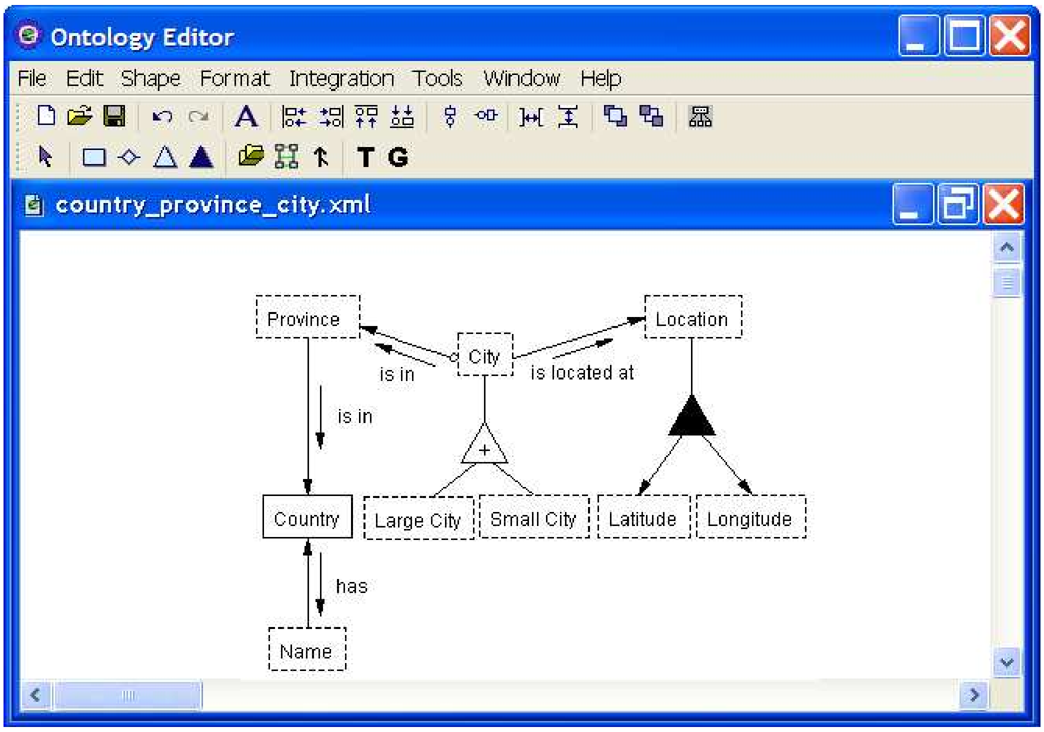


Figure 2.1: Sample Ontology Represented Graphically in the OntologyEditor

lationship set's name (e.g., *Province is in Country*). For n-ary relationship sets ($n > 2$), the relationship-set name must include names of the object sets for all of n connections, and must be fully written without using the reading-arrow shorthand. A small circle on one end of a relationship-set line indicates that an object's participation in a relation for the relationship set is optional. For example, in Figure 2.1 the optional constraint in relationship set *City is in Province* means that *City* can be recorded without associating it with a *Province* (e.g., Beijing is equivalent to a province in China's administrative structure, and it is not in any province). An arrowhead on a relationship-set line indicates that the relation is functional from domain (tail side) to range (head side). For example, in Figure 2.1 a *City* can have at most one *Location*, and therefore *City* functionally determines *Location*. In an ontology a *Generalization/Specialization* declares a superset/subset or "is-a" relationship between object sets. A triangle represents a generalization/specialization relationship. The apex of the triangle connects to (usually) one generalization object set and the base of the triangle connects to one or more specialization object sets. For example, in Figure 2.1 a *Large City* is a specialization of the generalization *City*. In an ontology an *Aggregation* declares a superpart/subpart relationship between object sets. A black triangle represents an aggregation. The apex of the triangle connects to the superpart and the base connects to the subparts. For example, in Figure 2.1 both *Longitude* and *Latitude* are subparts of a *Location*; together they constitute a location.

Now let us introduce the main functions on the OntologyEditor's toolbar as follows.

- "New" function. By clicking on 🗋, a user can create a new ontology.
- "Open" function. By clicking on 📂, a user can choose an ontology and load it into the OntologyEditor.
- "Save" function. By clicking on 💾, a user can save the opened ontology.
- "Select" function. By clicking on ↖, a user can select one or more ontology components.
- "New object set" function. By clicking on ▭, a user can create a new object set.
- "New relationship set" function. By clicking on ◇, a user can create a new relationship set.
- "New generalization/specialization" function. By clicking on △, a user can create a new generalization/specialization.
- "New aggregation" function. By clicking on ▲, a user can create a new aggregation.

The OntologyEditor, however, does not provide ontology mapping and merging functions. Our mapping and merging tool for TANGO has been built based on the OntologyEditor.

2.2 Loading Source and Target Ontologies

To do the ontology mapping and merging operations, users need two ontologies as input. Our tool designates one as the *target ontology* and the other as the *source ontology*. The target ontology is also called the *growing ontology*, which eventually grows to become the general domain ontology being created. The source ontology in the TANGO project is a lightweight mini-ontology generated according to a table. However, users actually can choose any ontology they want to integrate into the growing ontology.

The system provides an "open two ontologies" function, which can load the target and source ontologies together. A user first needs to open an ontology as in Figure 2.1. When the user then clicks on the "open two ontologies" button on the toolbar, the system designates the currently opene ontology as the target ontology and pops up a file-chooser dialog as in Figure 2.2 to let the user choose a source ontology. The user chooses a source ontology file and clicks on the open button. Figure 2.3 presents the two ontologies opened in one view. In this figure, the target ontology is on the left-hand side of the window, and the source ontology is on the right-hand side of the window.

The internal action of the "open two ontologies" function is as follows. When calling this function, the system automatically creates a new document with copies of the source and target ontologies. To accommodate them both in the same graphical interface, the system calculates a horizontal offset for the source ontology based on the target ontology's bounding-box size. With two ontologies open, users can still use all the normal OntologyEditor functions to modify components, add components, and save modified ontologies as needed to prepare ontologies for mapping and merging.

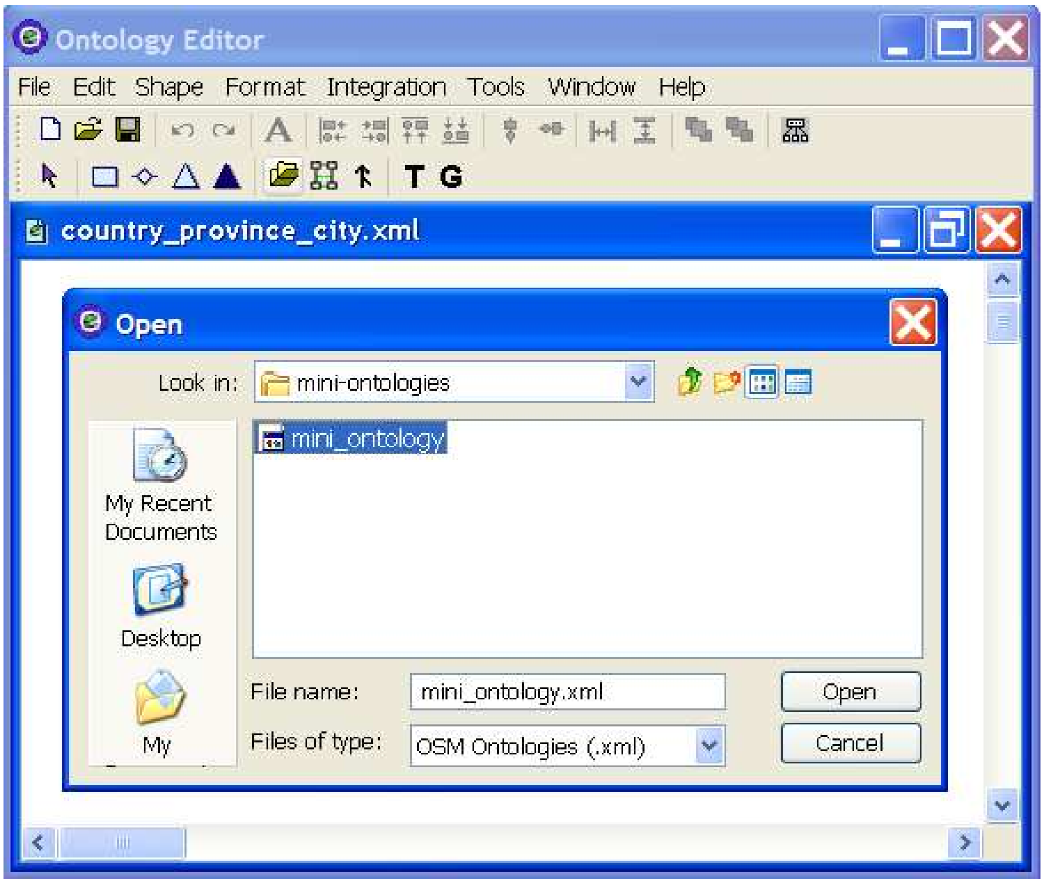


Figure 2.2: Choose Source Ontology Dialog

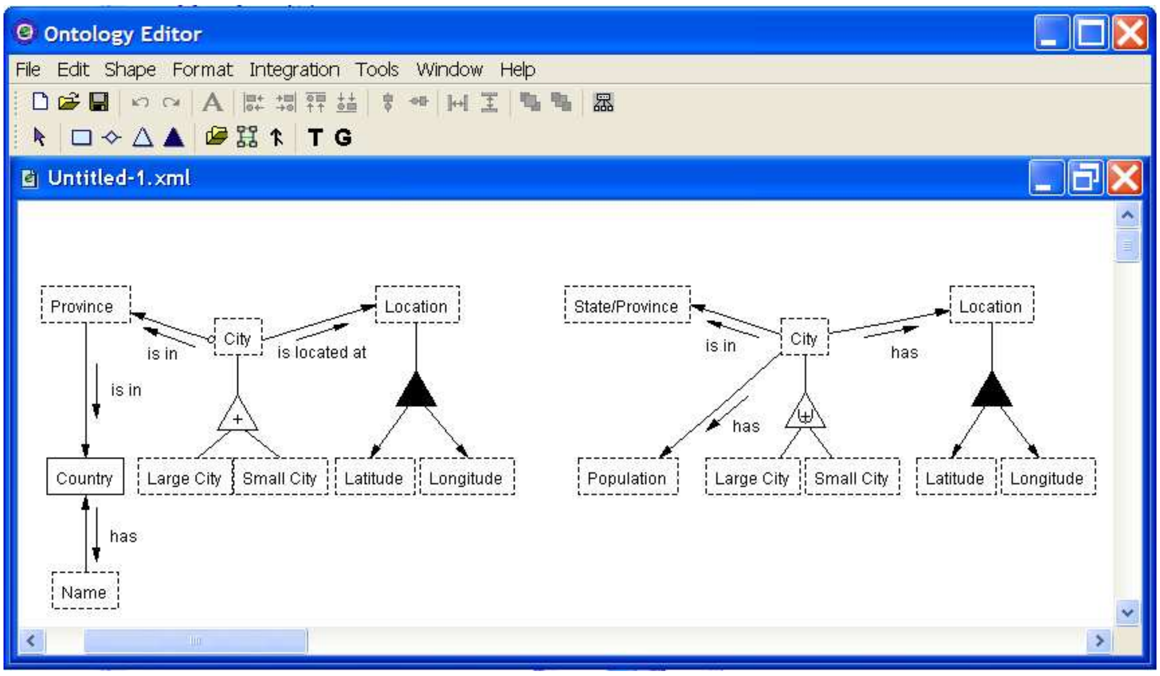


Figure 2.3: Target and Source Ontologies in the OntologyEditor

Chapter 3

Mapping and Merging

The system provides three modes for ontology mapping: manual, semi-automatic, and automatic. By clicking on the menu "Edit" and "Preference", a preference dialog window pops up. This dialog contains two user preference settings that users can change. In the "Choose Ontology Editor Type" panel, users can choose the ontology editor type: "Graphical Editor" or "Textual Editor". We discuss the graphical editor in Section 3.1.1 — Section 3.1.4 and discuss the textual editor in Section 3.1.5. In the "Choose the Mode" panel, users can choose "Manual Mode", "Semi-Automatic Mode" or "Automatic Mode". In the manual mode, users need to detect and create mappings manually. In the semi-automatic and automatic mode, a mapping algorithm helps users to find and create mappings. If there are any conflicts in the mappings, then in semi-automatic mode the system needs user interactions to resolve the conflicts, while in the automatic mode the system handles the mappings and resolves the conflicts automatically.

As the system operates, issues may arise in the ontology mapping and merging process. We use Issue/Default/Suggestion (IDS) statements [6] to handle these issues. An IDS statement raises an issue (I), specifies a default action (D) that it would carry out if the user does not intervene, and suggests (S) one or more alternative resolutions users may take.

To illustrate the mapping and merging processes, in this chapter we first introduce the manual mode in Section 3.1. We then introduce the semi-automatic and automatic mode in Section 3.2.

3.1 Manual Mode

The mapping button on the toolbar provides for the function of creating a mapping manually. A mapping can be created between two object sets, two relationship sets, two generalization/specializations or two aggregations. We only allow mappings between two components that have the same type (i.e., object sets can only map to object sets, relationship sets can only map to relationship sets, etc.). Since lexical and nonlexical object sets are different types of concepts, we do not allow a mapping between them either. The two mapped components have to be from different ontologies — our system does not allow a mapping between two components in the same ontology.

The basic operation for creating mappings between two components has three steps. (1) A user clicks on the mapping button to start a mapping operation. (2) The user chooses and clicks on one component to be mapped from either the source ontology or the target ontology. (3) The user drags the mouse cursor to the other component to be mapped and releases the mouse button.

The system provides a user-friendly graphical interface that lets users know if an operation is appropriate: when dragging the mouse cursor, a dashed line appears to indicate the potential mapping. If the mouse hovers over an invalid ontology component for the mapping, an invalid icon 🚫 appears. If the mouse hovers over a valid component for the mapping, the tool shows a green arrow to indicate that the mapping can be created. When the user releases the mouse button hovering over a valid component, the mapping is created.

Sometimes a mapping might cause one or more resolvable conflicts. If a conflict occurs, IDS interactions can help users handle it. In the following subsections, we discuss conflicts and their corresponding IDSs in mappings between object sets, relationship sets, generalization/specializations, and aggregations.

3.1.1 Object Set Mapping

Some mappings between two object sets are simple. Users can simply connect the two object sets to indicate they map to each other, and the mapping is complete.

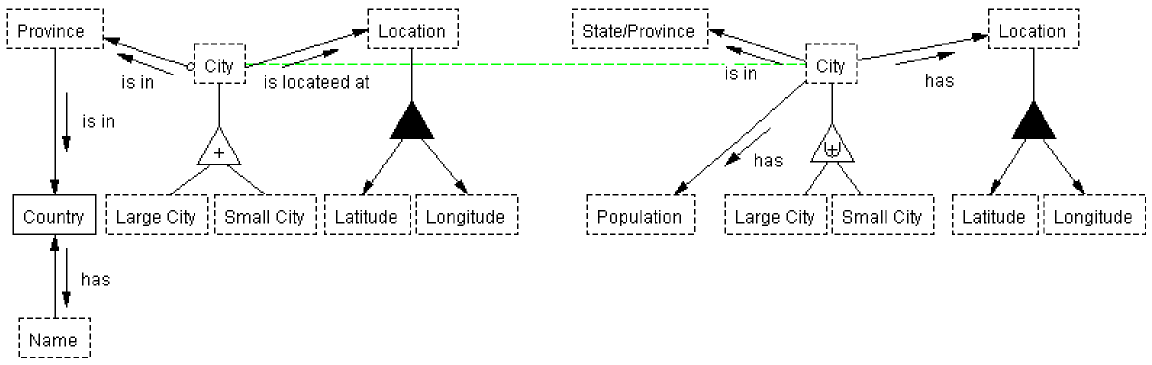
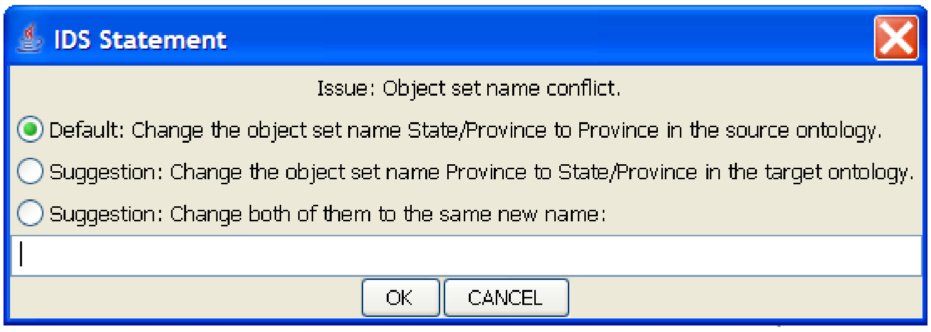


Figure 3.1: A Mapping Between Two Object Sets

For example, the dashed line between the two object sets *City* in Figure 3.1 indicates that these two objects map to each other. The component types match and the names are identical, so the system accepts the mapping as specified.

Sometimes two object sets may have the same semantic meaning, but their names are different. When a user creates a mapping between them, the system raises an IDS statement. For example, in Figure 3.1, the object set *Province* in the target ontology and the object set *State/Province* in the source ontology are considered by the users to be semantically equivalent. After a user creates a mapping between them, an IDS statement is raised because they do not have the same name. Figure 3.2 shows the IDS statement. In an IDS pop-up window, the system lists the issue "Object set name conflict", a default solution "Change the object set name State/Province to Province in the source ontology" and two suggested solutions "Change the object set name Province to State/Province in target ontology" and "Change both of them to the same new name" followed by a text box. Assuming the user wishes to choose the first suggestion, the user click on the radio button for the suggestion and then clicks on the OK button. Figure 3.3 shows the result. In this case, the object set *Province* in target ontology changed to *State/Province*, and a dashed line representing the mapping relation between the two object sets appears. If the user had chosen the default, the system would have made both names be *Province*. If the user had chosen


IDS Statement

Issue: Object set name conflict.

[✓ Default: Change the object set name State/Province to Province in the source ontology.]

Suggestion: Change the object set name Province to State/Province in the target ontology.

Suggestion: Change both of them to the same new name:

Figure 3.2: An IDS Statement for Object Set Name Conflict

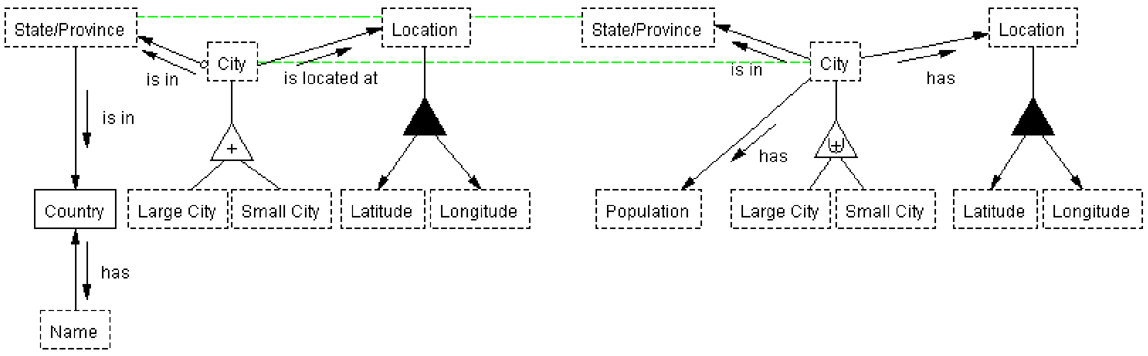


Figure 3.3: Another Mapping Between Two Object Sets

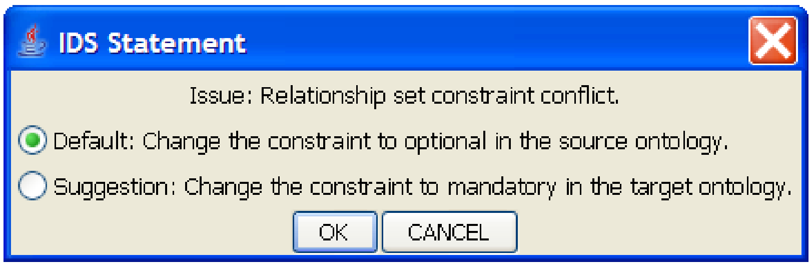
the second suggestion, the system would have changed the names of both object sets to the new name given by the user.

3.1.2 Relationship Set Mapping

To match two relationship sets, users can simply draw a line between the two relationship sets. If the connecting object sets have already been mapped, the names are the same, and the constraints on the relationship set are the same, the system accepts the mapping.

For conflicts between relationship sets the system provides two IDS statements. One is the constraint-conflict IDS statement; the other is the relationship-set-name-conflict IDS statement.

In Figure 3.3, both the target ontology and the source ontology have a relationship set *City is in State/Province*. Their constraints, however, are different.


IDS Statement

Issue: Relationship set constraint conflict.

[✓ Default: Change the constraint to optional in the source ontology.]

Suggestion: Change the constraint to mandatory in the target ontology.

Figure 3.4: An IDS Statement for Relationship Sets Constraint Conflict

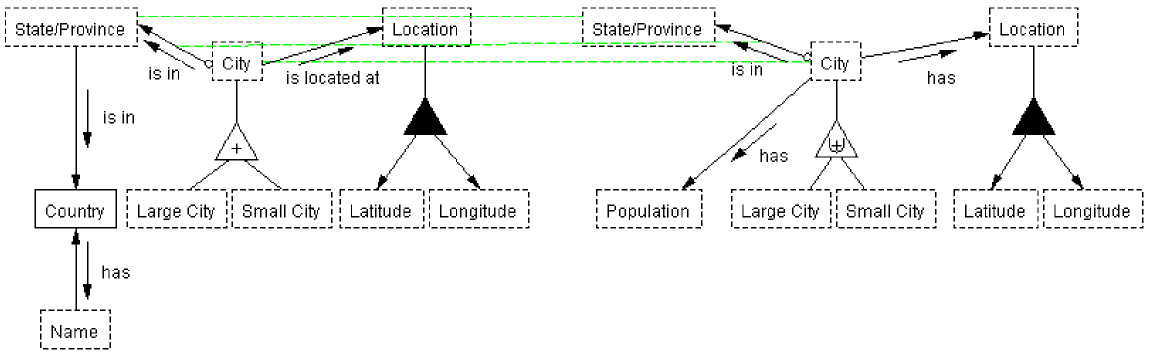
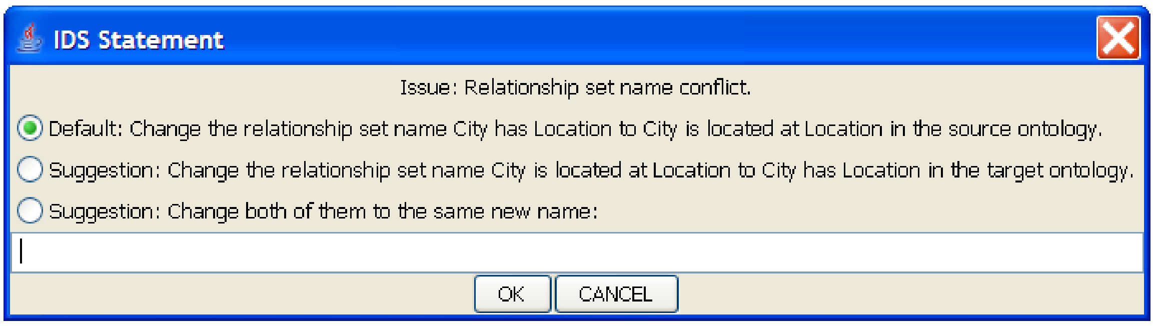


Figure 3.5: A Mapping Between Two Relationship Sets

In the target ontology, the relationship set has both a functional and an optional constraint. In the source ontology, the corresponding relationship set only has the functional constraint. When a user creates a mapping between these two relationship sets, the IDS statement in Figure 3.4 pops up. This IDS statement lists the issue "Relationship set constraint conflict", the default solution "Change the constraint to optional in the source ontology", the a suggested solution "Change the constraint to mandatory in the target ontology". Assuming the user chooses the default solution and clicks on the OK button, Figure 3.5 represents the mapping result. If the user chooses the suggestion, the system removes the optional constraint in the target ontology, making the participation of the *City* object set mandatory.

If two relationship sets are to be matched, all the object sets involved in the relationship sets have to be matched. Our system provides an "all but one" function to make the operation simpler. If all but one of the object sets for mapping between a relationship set match, the system knows to match the remaining object sets — one



IDS Statement

Issue: Relationship set name conflict.

[✓ Default: Change the relationship set name City has Location to City is located at Location in the source ontology.]

Suggestion: Change the relationship set name City is located at Location to City has Location in the target ontology.

Suggestion: Change both of them to the same new name:

Figure 3.6: An IDS statement for Relationship Set Name Conflict

for the source to one for the target. For example, in the two ontologies in Figure 3.5, we first see that the relationship set *City is located in Location* in the target ontology and the relationship set *City has Location* in the source ontology have the same semantic meaning, though their relationship-set names are different. When creating the mapping, the IDS statement in Figure 3.6 appears. It lists the issue “Relationship set name conflict”, the default solution “Change the relationship set name City has Location to City is located at Location in the source ontology” and two suggested solutions: “Change the relationship set name City is located at Location to City has Location in the target ontology”, and “Change both of them to the same new name” followed by a text box. Assuming the user chooses the first suggestion, the system changes the relationship set name *City is located at Location* to *City has Location* in the target ontology and generates the new mapping. The object sets *Location* in the target and source ontologies also automatically map together with the “all but one” resolution. Figure 3.7 shows the result.

3.1.3 Generalization/Specialization Mapping

The process of mapping two generalization/specializations is similar to the process of declaring a mapping between two relationship sets. Users just need to connect the two generalization/specializations to create the mapping.

There are constraint conflicts in generalization/specializations mappings too. A generalization/specialization relationship has four types of constraints: Mutual

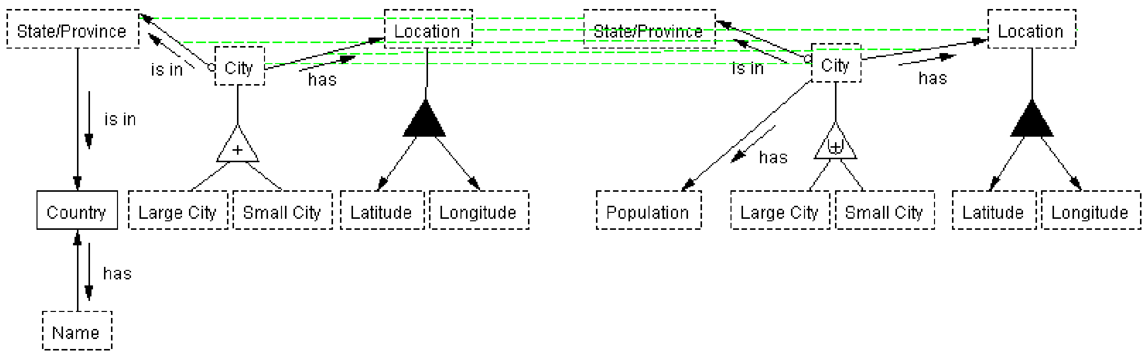
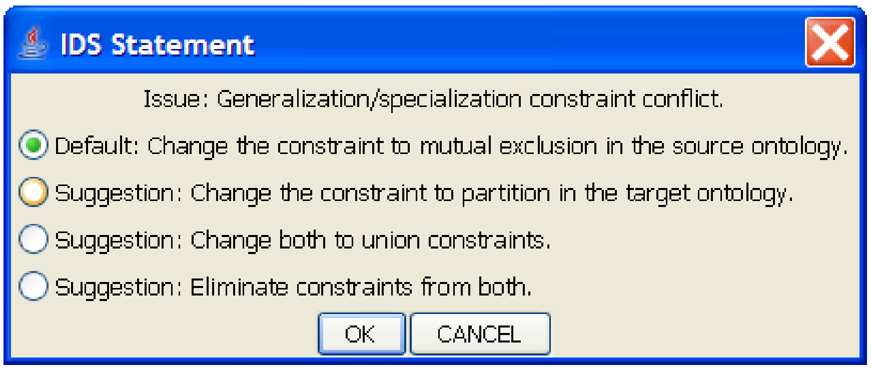


Figure 3.7: Another Mapping Between Two Relationship Sets

exclusion (Mutex), Partition, Union, and Intersection. If we map two generalization/specializations with different constraints, an IDS statement is raised. For example, in Figure 3.7, the generalization/specialization in the target ontology has a mutual exclusion constraint, and the generalization/specialization in the source ontology has a partition constraint. When a user maps them, a constraint conflict is raised. Figure 3.8 shows the IDS statement for this generalization/specialization-constraint conflict. It addresses the issue "Generalization/specialization constraint conflict", offers the default resolution, "Change the constraint to mutual exclusion in the source ontology" and three suggestions: "Change the constraint to partition in the target ontology", "Change both to union constraints", and "Eliminate constraints from both". Assuming the user chooses the default solution, the system then changes the partition constraint in the source ontology to a mutual exclusion constraint and creates the mapping.

Both of these generalization/specializations have object sets *Large City* and *Small City* on their specialization sides. They have the identical names, but they may have different semantic meanings. For instance, in the target ontology a large city can be defined as a city with a population that is more than 10 million, while in the source ontology the object set *Large City* can be defined as a city with a population that is more than 5 million. The user must decide whether they should be matched. In our example, we assume that the user knows that the object sets *Small City* in both


IDS Statement

Issue: Generalization/specialization constraint conflict.

[● Default: Change the constraint to mutual exclusion in the source ontology.]

○ Suggestion: Change the constraint to partition in the target ontology.

○ Suggestion: Change both to union constraints.

○ Suggestion: Eliminate constraints from both.

OK CANCEL

Figure 3.8: An IDS statement for Generalization/Specialization Constraint Conflict

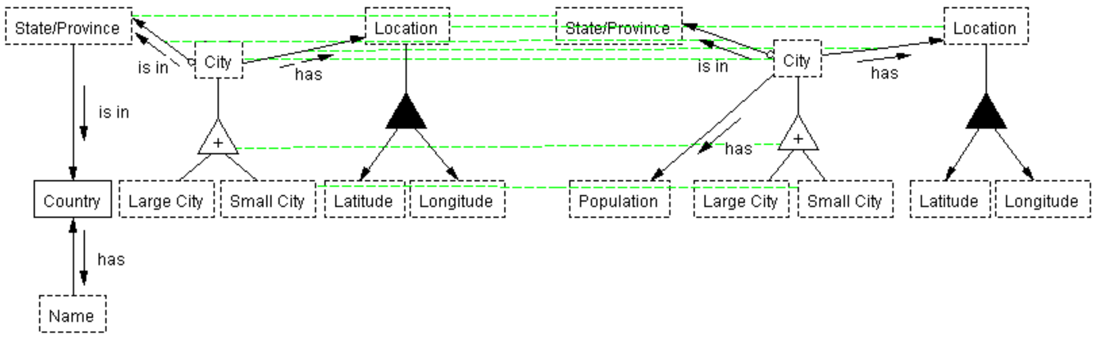


Figure 3.9: A Mapping Between Two Generalization/Specialization Relationships

the target ontology and the source ontology are defined as a city with a population that is less than 5 millon. Therefore, the two object sets *Small City* are semantically equivalent, but the two object sets *Large City* are not. The user then needs to create a mapping between the object sets *Small City*. Figure 3.9 represents the result.

3.1.4 Aggregation Mapping

The process of mapping two aggregations is similar to the process of creating a mapping between two generalization/specializations. Users can indicate that there should be a mapping by connecting two aggregations.

An aggregation can only have one superpart. Thus for a pair of mapped aggregations, their superparts have to be mapped. When a user maps two aggregations, the system automatically verifies whether the two superparts are mapped. If they are

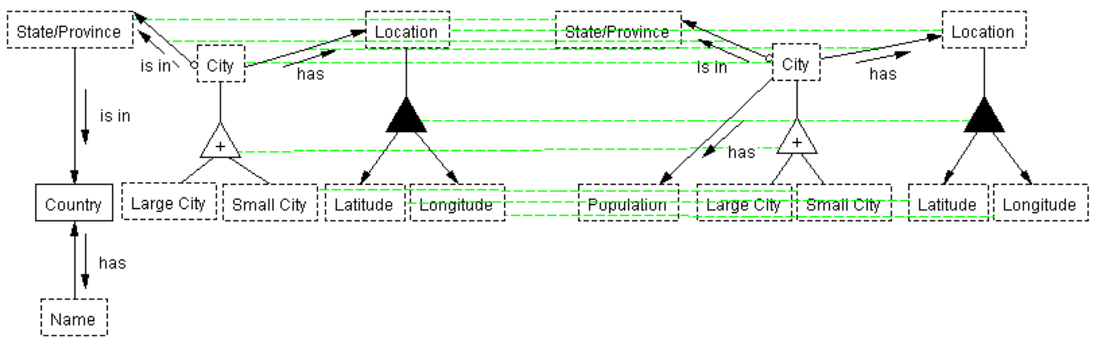


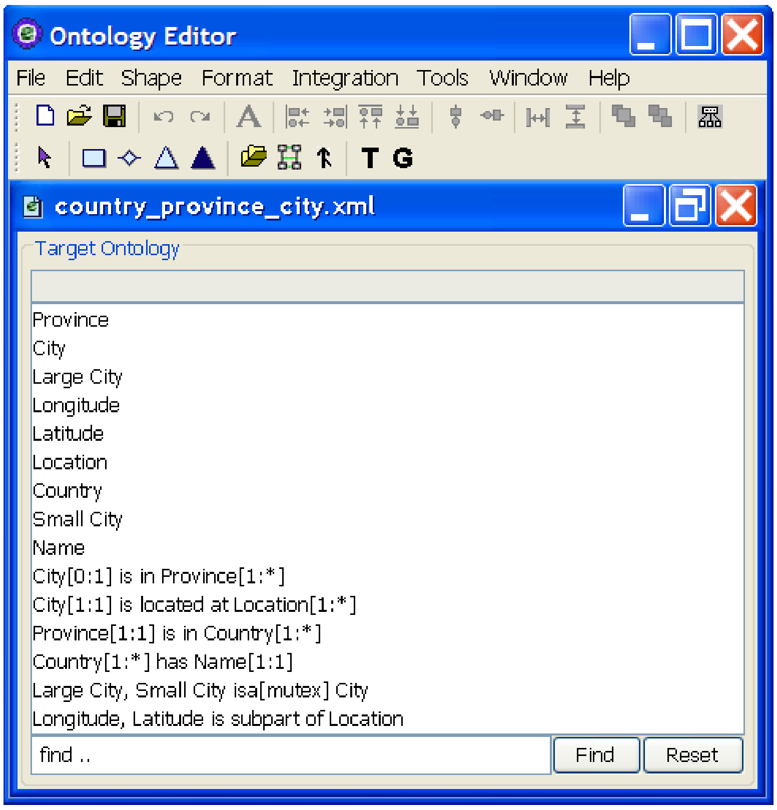
Figure 3.10: A Mapping Between Two Aggregations

not mapped and their names are the same, the system automatically creates a mapping between them. If they are not mapped but their names are different, an object set name conflict IDS statement appears to help the user create the mapping between these object sets. The aggregations *Latitude, Longitude is subpart of Location* in the target and source ontologies represent the same semantic meaning. *Latitude* in the target and source ontologies have the same name and same semantic meaning, — similarly for *Longitude*. Therefore, the user can create mappings for these two sets of object sets. Figure 3.10 shows the mapping result.

3.1.5 Text Editor View

As ontologies grow large, it becomes unwieldy to manually specify mappings in the graphical view we have been discussing. We therefore provide another user interface, a text view, in which ontology components are described textually. In Figure 4.2, in the "Choose Ontology Editor Type" panel, if a user chooses *"Textual Editor"*, the system sets it as the user-preferred editor. Then when opening an ontology, the system opens it in the text editor view instead of in the graphical editor view.

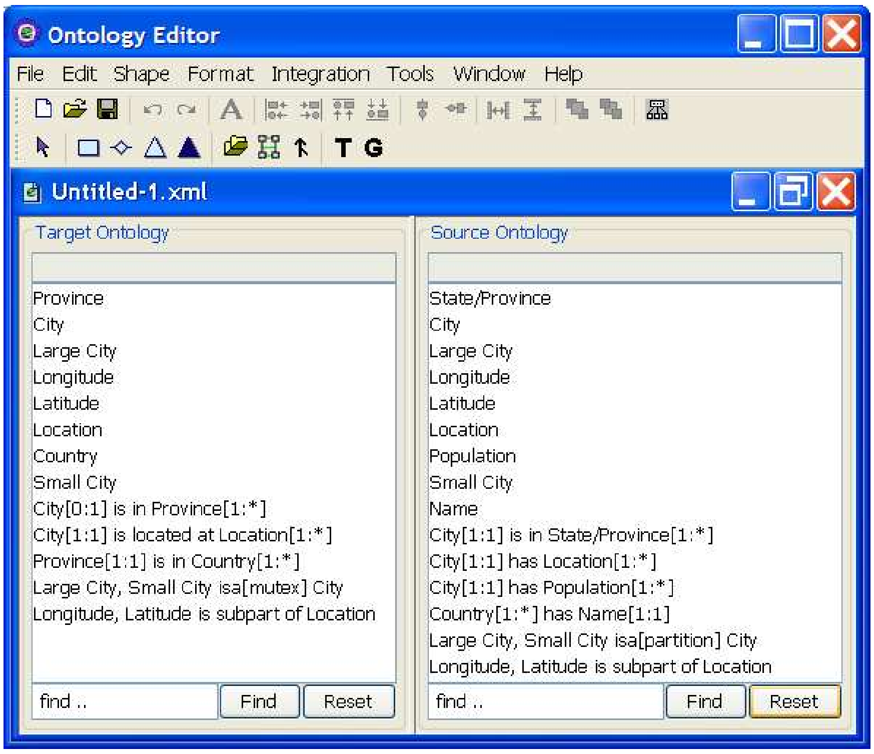
Figure 3.11 shows the target ontology from Figure 2.1 in the text editor view. The textual language is OSM-L[10, 16]. The textual view lists the object sets *Province*, *City*, *Large City* etc.; the relationship sets *City[0:1] is in Province[1:*]*, *City[1:1] is located at Location[1:*]*, etc.; the generalization/specialization *Large City,* *Small City isa [mutex] City*; and the aggregation *Longitude, Latitude is subpart of Location*. The system automatically transforms the optional and functional constraints to participation constraints. For example, *City[0:1] is in Province[1:*]* means that each *City* object is associated with zero or one *Province* objects and each *Province* object is associated with one or more *City* objects.


Ontology Editor
File Edit Shape Format Integration Tools Window Help
T G
country_province_city.xml
Target Ontology

Province
City
Large City
Longitude
Latitude
Location
Country
Small City
Name
City[0:1] is in Province[1:*]
City[1:1] is located at Location[1:*]
Province[1:1] is in Country[1:*]
Country[1:*] has Name[1:1]
Large City, Small City isa[mutex] City
Longitude, Latitude is subpart of Location

Figure 3.11: The Target Ontology in Text View Editor

After clicking on the "open two ontologies" button and choosing the source ontology, the system loads them together as Figure 3.12 shows. As in the graphical editor, the target ontology is on the left hand side and the source ontology is on the right hand side. In the manual mapping mode, users can choose mapping candidates by clicking on the ontology components on the target and source ontology lists. In Figure 3.14, the highlighted component *Province* in the target ontology and



Target Ontology:

Province
City
Large City
Longitude
Latitude
Location
Country
Small City
City[0:1] is in Province[1:*]
City[1:1] is located at Location[1:*]
Province[1:1] is in Country[1:*]
Large City, Small City isa[mutex] City
Longitude, Latitude is subpart of Location

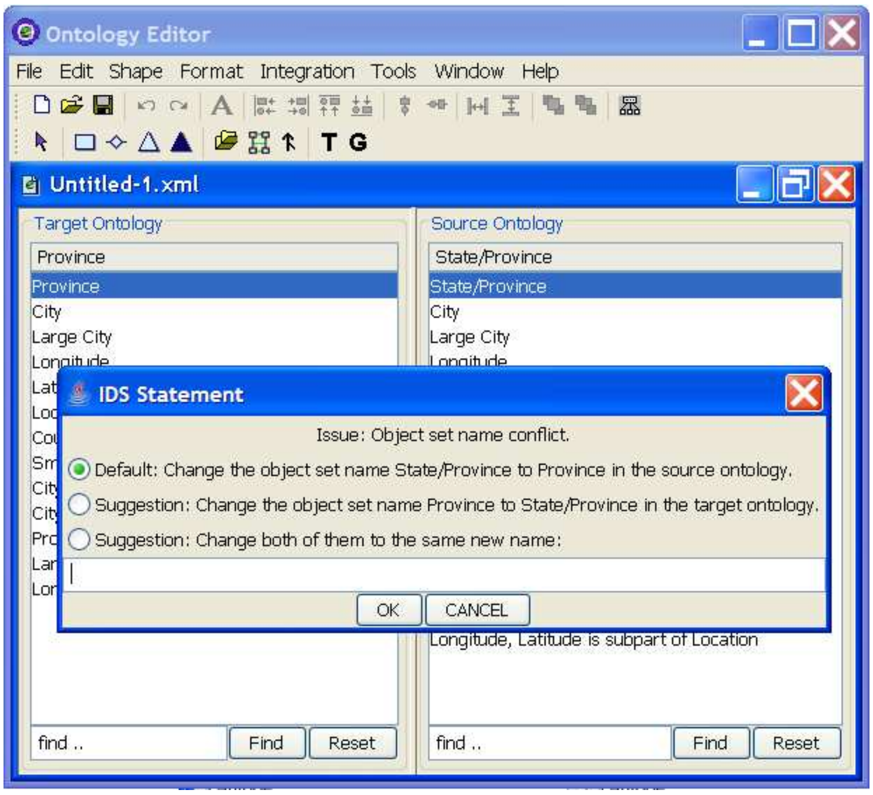
Source Ontology:

State/Province
City
Large City
Longitude
Latitude
Location
Population
Small City
Name
City[1:1] is in State/Province[1:*]
City[1:1] has Location[1:*]
City[1:1] has Population[1:*]
Country[1:*] has Name[1:1]
Large City, Small City isa[partition] City
Longitude, Latitude is subpart of Location

Figure 3.12: Open Two Ontologies in Text View Editor

State/Province in the source ontology has been selected and the user has clicked on the mapping button . As is the case for the graphical editor, if no IDS conflict is raised, a mapping is created. Otherwise, an IDS statement pops up. In this example, the system pops up an IDS statement as Figure 3.13 shows. If the user chooses the first suggestion as the resolution, the system changes the object set in the source ontology and creates a mapping. To record the mapping, a mapping panel lists the result as Figure 3.14 shows. It lists the mapping of object sets *State/Province*.

A user may want to view target and source ontologies in both the graphical view and the textual view. To provide this option, the system provides "show in text editor view" and "show in graphical editor view" functions. If a user has an open ontology in graphical editor view, by clicking on the "show in text view editor" button **T** on the toolbar, a text editor view of this ontology appears. Similarly, if the user has open text editor view, by clicking on **G**, a graphical editor view appears. In either case, the graphical editor and text editor view share the same ontology model.



IDS Statement

Issue: Object set name conflict.

[✓ Default: Change the object set name State/Province to Province in the source ontology.]

Suggestion: Change the object set name Province to State/Province in the target ontology.

Suggestion: Change both of them to the same new name:

Target Ontology	Source Ontology
Province	State/Province
City	City
Large City	Large City
Longitude	Longitude

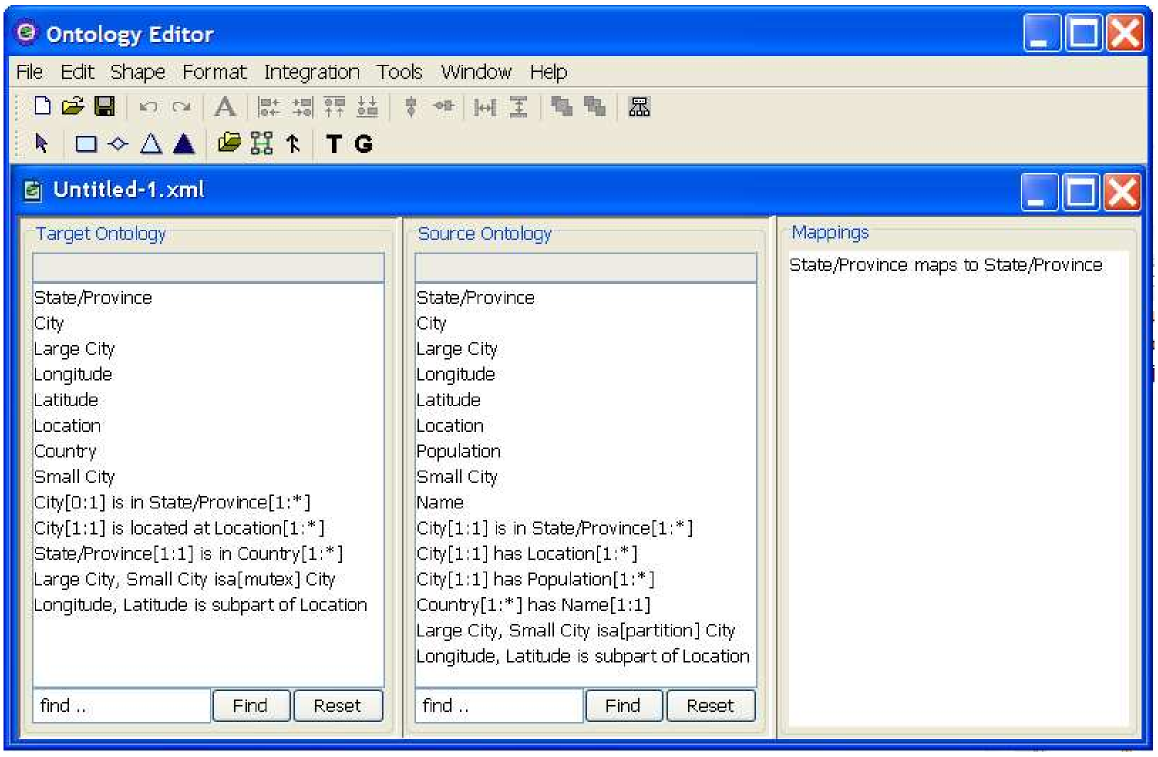
Longitude, Latitude is subpart of Location

Figure 3.13: An IDS Statement for Object Set Name Conflict in the Text View Editor

If the user makes any change using either editor, the change automatically appears in the other editor.

3.2 Semi-Automatic and Automatic Mode

In the semi-automatic mode and automatic mode, the system employs a mapping algorithm to detect and generate mappings. We provide an API for mapping algorithms that allows the system and the mapping algorithms to work with each other. (Chapter 4 describes the API.) The system provides the target and source ontologies (both usually populated) to a mapping algorithm. Using this information, a mapping algorithm discovers and creates mappings and then sends them back to the system. The system uses IDS statement processing to validate the mappings. In the semi-automatic mode, if the mappings contain conflicts, the system pops up IDS statements to users and resolves the conflicts based on user interactions. In the



Target Ontology

State/Province
City
Large City
Longitude
Latitude
Location
Country
Small City
City[0:1] is in State/Province[1:*]
City[1:1] is located at Location[1:*]
State/Province[1:1] is in Country[1:*]
Large City, Small City isa[mutex] City
Longitude, Latitude is subpart of Location

find .. Find Reset

Source Ontology

State/Province
City
Large City
Longitude
Latitude
Location
Population
Small City
Name
City[1:1] is in State/Province[1:*]
City[1:1] has Location[1:*]
City[1:1] has Population[1:*]
Country[1:*] has Name[1:1]
Large City, Small City isa[partition] City
Longitude, Latitude is subpart of Location

find .. Find Reset

Mappings

State/Province maps to State/Province

Figure 3.14: Two Mapped Ontologies in the Text View Editor

automatic mode, the system automatically uses the default resolution to resolve the conflicts and user interactions are not necessary.

As an example, we have provided a naive mapping algorithm. Since relationship sets, generalization/specializations and aggregations are based on object sets, the system starts with object set mappings first. It then works on relationship set mappings, generalization/specialization mappings and aggregation mappings. Figure 3.15 shows the mapping conditions of the naive mapping algorithm. Basically, the algorithm naively creates mappings between ontology components according to their names.

Consider our running example to illustrate the semi-automatic process. In the semi-automatic mode, when the user clicks on the mapping button, the mapping algorithm can detects, creates, and returns mapped ontologies to the system. In our example, the system finds a relationship set name conflict and a generalizatiion/specialization constraint conflict as the IDS statements in Figure 3.6 and

Component Name	Mapping Condition
Object sets	Same name
Relationship sets	All object sets in a relationship set are mapped.
Generalization/specialization	At least a pair of object sets on the generalization side is mapped. At least a pair of object sets on the specialization side is mapped.
Aggregation	The object sets on the super-part side are mapped. At least a pair of object sets on the subpart side is mapped.

Figure 3.15: The Native Mapping Algorithm Mapping Conditions

Figure 3.8 show. After the IDS statements pop up, the user needs to select the resolutions for the conflicts. Assuming that the user chooses the same resolutions as discussed earlier, Figure 3.16 shows the result. Since the naive mapping algorithm is simply based on the names, it does not consider additional information about the semantic meaning of the ontology components. It maps the ontology components with the same name even if they have different semantic meanings (e.g., object sets *Large City* in the target and source ontologies) and it cannot detect mappings between two ontology components that have the same semantic meaning but have different names (e.g., the object set *Province* and *State/Province*, and relationship sets *City is in Province* and *City is in State/Province*). Before merging the mapped ontologies, users can modify the mapping results. In this example, the user can manually add the mappings between *Province* and *State/Province* and also the mappings between the relationship sets *City is in Province* and *City is in State/Provice.* To remove a mapping between two ontology components, the user clicks on the select button on the toolbar, selects the mapping to be removed, and then presses the delete key on the keyboard. In our example, the user needs to remove the mapping between the two object sets *Large City.* With these actions the user will have produced the mapping in Figure 3.10 and will have produce it with less effort.

In the automatic mode, the system automatically resolves mapping conflicts with the default resolutions in the IDS statements. Figure 3.17 shows the mapping

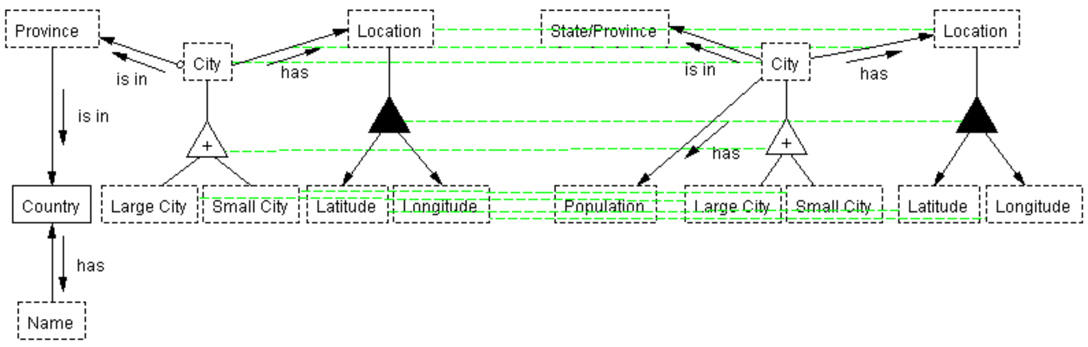


Figure 3.16: The Naive Mapping Algorithm Generated Mappings in Semi-Automatic Mode

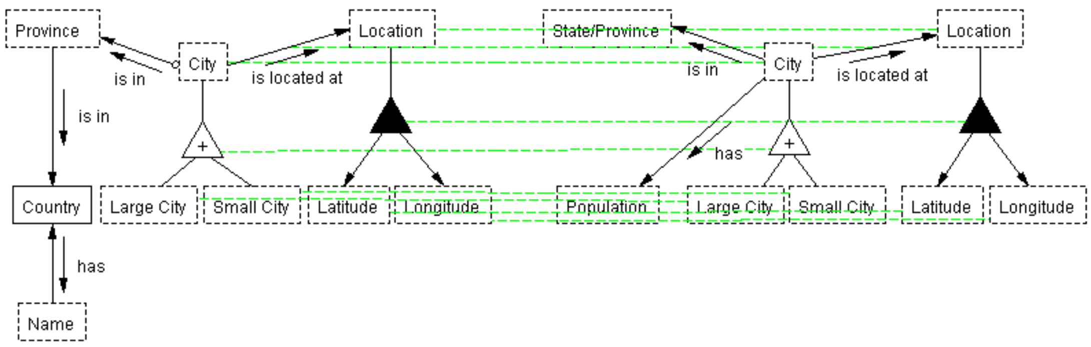
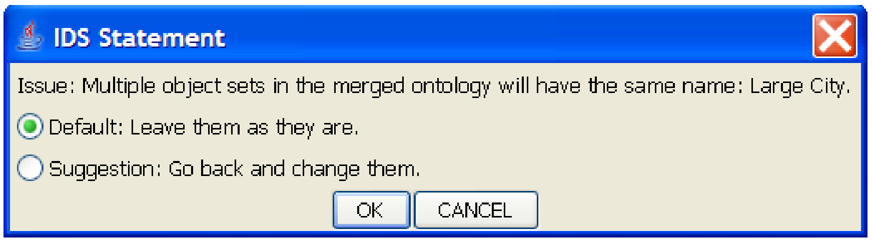


Figure 3.17: The Naive Mapping Algorithm Generated Mappings in Automatic Mode

result of our running example in automatic mode. In this figure the relationship set between *City* and *Location* is *City is located at Location*, because this is the default resolution for the relationship set name conflict in Figure 3.6. As in the semi-automatic mode, users can review the result and add/remove mappings before they launch the merging process.

3.3 Merge Ontologies

To merge mapped target and source ontologies, a user can click on the merge button. If multiple unmapped object sets have the same name, our system provides an IDS statement to remind the user to make them different. Consider the mapped ontologies in Figure 3.10. In this example, both the target and source ontologies have object sets *Large City*. As we discussed in the previous section, their



IDS Statement

Issue: Multiple object sets in the merged ontology will have the same name: Large City.

[✓ Default: Leave them as they are.]

Suggestion: Go back and change them.

Figure 3.18: An IDS for Multiple Object Sets Have the Same Name

semantic meanings are different. Figure 3.18 shows the IDS pop-up window. It addresses the issue "Multiple object sets in the merged ontology will have the same name Large City", provides the default resolution "Leave them as they are" and the suggestion "Go back and change them". If the user chooses the default resolution, the system keeps their names and processes the merge. If the user chooses the suggestion, the system will cancel the merge operation and go back to the editor window. The user can then change the object set names. Assuming the user chooses the default resolution, the system creates a new ontology and merges the mapped object sets, relationship sets, generalization/specializations, and aggregations. For the unmapped components — in our example *Country*, *Name*, *Large City*, *Country has Name*, and *State/Province is in Country* — our system automatically copies them to the merged growing ontology.

Often in this merged ontology the components' positions are not organized very well. In the graphical view, ontology components have attributes giving their x and y coordinates. In the merged ontology, these attribute values are from either the target ontology or source ontology. They may not work well for the new merged ontology. Figure 3.19 shows the resulting merged diagram. The user may, of course, rearrange the diagram to make its appearance tidy. Alternatively, the OntologyEditor provides an automatic layout function. By clicking on the layout button , the system can automatically re-arrange the ontology components' positions. Figure 3.20 shows the merged ontology whose layout is arranged by the layout function. Again, the user may adjust the diagram to make its appearance tidy.

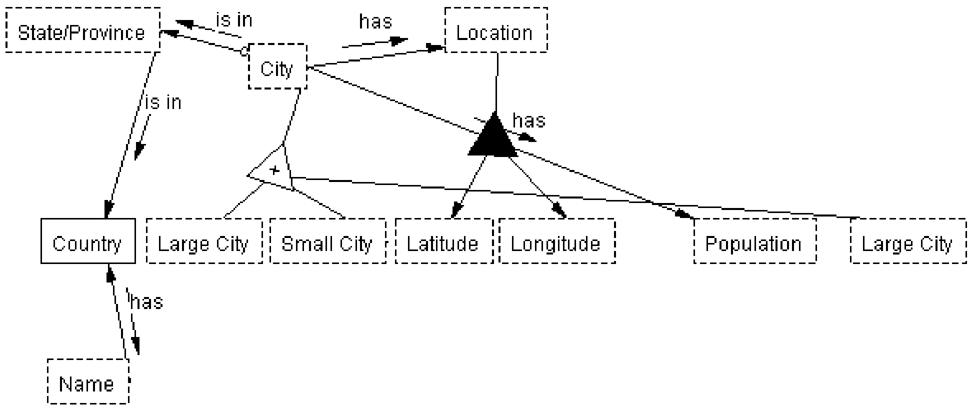


Figure 3.19: The New Growing Ontology After Merging

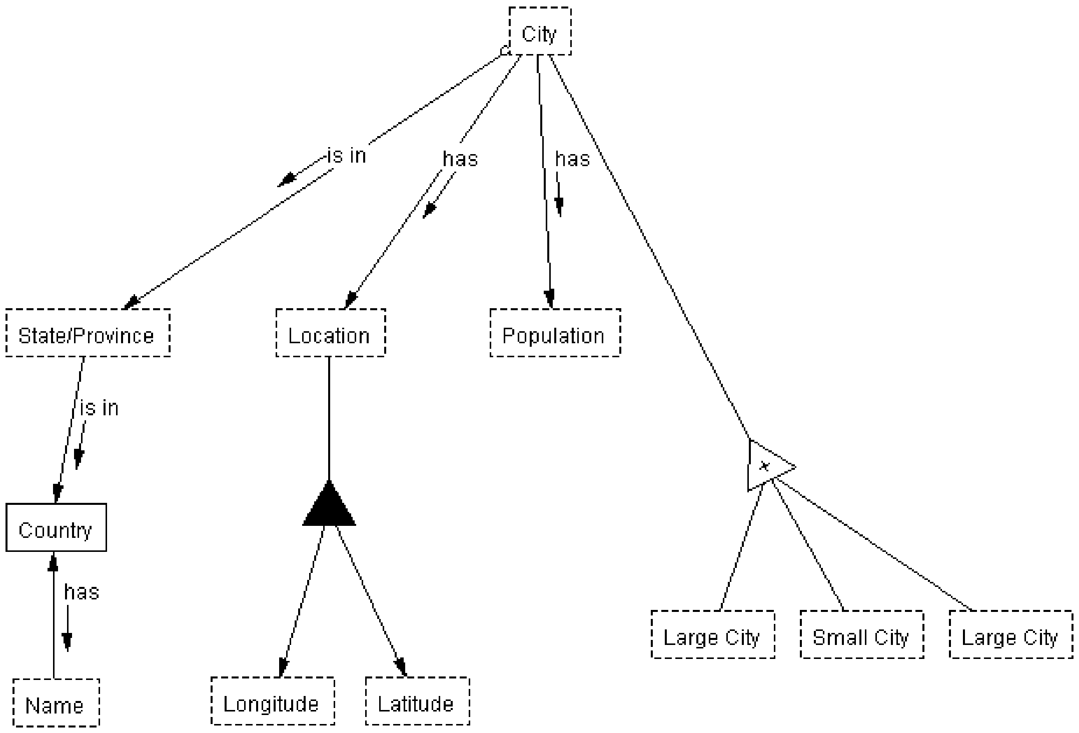


Figure 3.20: The New Growing Ontology After Applying the Layout Function

Chapter 4

API for Plug-in Mapping and Merging Algorithms

The system provides an API (Application Programming Interface) for developers to plug in mapping and merging algorithms. In this chapter, we first introduce the API, and then explain how to plug in an algorithm to the system. After a mapping algorithm is plugged in, the system can perform automatic and semi-automatic ontology mappings. For merging, the API provides the functions necessary to merge the data in populated ontologies.

The full API description is in Appendix A. Here we give a general description of the API and how it is used for the provided plug-in mapping and merging algorithms.

The system provides a Java interface class *Algorithm* for mapping and merging algorithms. It consists of one method: *run(OntologyModel m)*. This method makes the target and source ontologies accessible to the developer's algorithm. A developer must implement this interface class when designing a new mapping or merging algorithm.

Based on the *OntologyModel* class, the methods in the API allow developers to obtain and provide information about the source and target ontologies as well as the mappings between them and the data instances stored in them. The *OntologyModel* class provides a set of methods to access the ontology components in the target and source ontologies (e.g., *getTargetOntologyObjectSetList*, *getSourceOntologyObjectSetList*, *getTargetOntologyRelationshipSetList*, *getSourceOntologyRelationshipSetList*, etc.), a set of methods to create different ontology component (e.g., *createAggregationInTargetOntology*, *createAggregatiionInSourceOntology*, *createObjectSetInTargetOntology*, *createObjectSetInSourceOntology*, *createMapping*, etc.), and the method *remove* to delete ontology components. The *OntologyModel* also provides the method

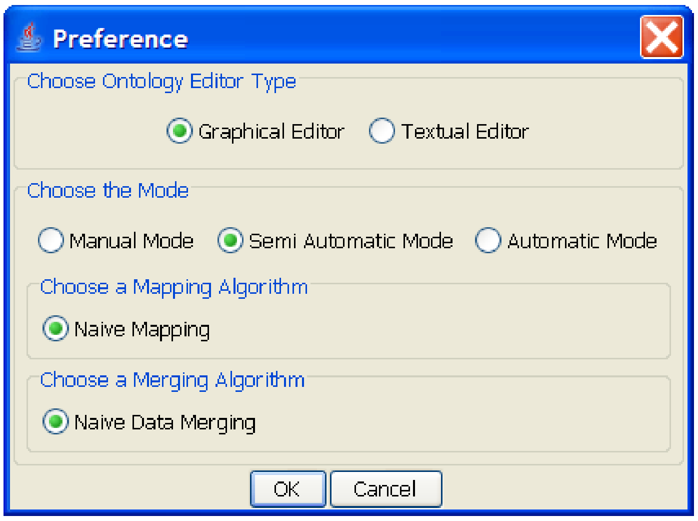
```
<PlugIns>
    <MappingPlugIn
        class="edu.byu.deg.plugin.algorithms.NaiveMapping"
        name="Naive Mapping"
        description="Detect mappings according to the object set name"
    />
    <DataMergePlugIn
        class="edu.byu.deg.plugin.algorithms.NaiveDataMerging"
        name="Naive Data Merging"
        description="Merge the data instances assuming no objects or relationship are the same"
    />
< /PlugIns>
```

Figure 4.1: Registration for Mapping and Merging

createMapping, and to delete mappings, it provides the method *removeMapping.* To access instances, the API provides different methods according to the different ontology components. For example, an object set can access its data instances with the method *getObjectList.* Finally, to merge data, the *OntologyModel* provides the methods *sameObjectAs* and *sameRelationshipAs*, and to delete object and relationship mappings, it provides the methods *removeSameAsMapping.*

To register an algorithm with the system, a developer must add a "plug-in entry" to the XML file PluginAlgorithms.xml that lists the "plug-ins" for the system. Figure 4.1 shows an example. Two types of plug-ins are allowed: The"MappingPlugIn" is for mapping algorithms, and the "DataMergePlugIn" is for data merging algorithms. A plug-in entry for a mapping algorithm has three attributes: *class* specifies the algorithm class name; *name* specifies the algorithm's name, which the system displays for possible use by the end user; and *description* adds comments about the algorithm.

After registering an algorithm to the "PluginAlgorithms.xml" file, the algorithm's classes can be loaded in the system. A user then can select the plugged-in algorithm with the preference setting as Figure 4.2 shows. In the preference window, when users choose to use the semi-automatic or automatic mode, they need to choose their preferred mapping and data merging algorithms. For the initial system only one mapping algorithm and one data merge algorithm are available. We described



Preference

Choose Ontology Editor Type

[✓ Graphical Editor] Textual Editor

Choose the Mode

Manual Mode [✓ Semi Automatic Mode] Automatic Mode

Choose a Mapping Algorithm

[✓ Naive Mapping]

Choose a Merging Algorithm

[✓ Naive Data Merging]

Figure 4.2: The Preference Window for Setting the Mapping and Data Merging Algorithms

the *Naive Mapping* algorithm in Chapter 3. The *Naive Data Merging* algorithm is an algorithm that preserves all data assuming no objects or relationships are the same.

Chapter 5

Observations and Analyses

As a part of the TANGO project and for the purpose of this thesis, we tested the tool on the geopolitical domain, where relevant empirical data is widely scattered but often presented in the form of tables. Based on these tables, the TANGO system can generate a set of mini-ontologies (see Chapter 1). We used these mini-ontologies as the test cases. In this chapter, we describe our experience and report our observations. We also discuss the strengths and weaknesses of our tool.

5.1 Preparation

To test our tool, we chose 12 tables from the geopolitical domain and converted each one, with the OntologyEditor, to mini-ontologies. These 12 mini-ontologies are in Appendix B. These 12 mini-ontologies contain 55 object sets and 43 relationship sets. Using these mini-ontologies as test cases, an expert tested the system using four different methods to perform the ontology integration: (1) using the OntologyEditor to create the domain ontology; (2) using the manual mode of our tool to manually map and merge the mini-ontologies; (3) using the semi-automatic mode of our tool to map and merge the mini-ontologies; and (4) using the automatic mode of our tool to map and merge the mini-ontologies. To obtain rough estimates of the effort required to complete each task, we timed the mapping and merging performance of the expert user for each method.

5.2 Results and Observations

Figure 5.1 shows the merged domain ontology, which contains 40 object sets and 39 relationship sets. There were 20 mappings, and 18 conflict issues were raised in these mappings.

As an example of the process, Figure 5.2 shows how a growing ontology (on the left) maps with the mini-ontology in Appendix B, Figure B.4 (on the right). When the expert created these mappings, two object set name conflicts and a relationship set participation constraint conflict were raised. The original object set name for *Country* in the source ontology was *Country or Region* (see Figure B.4). The expert resolved the conflict by choosing the default and the system changed the name to *Country*. Similarly, the *Name* object set which in the source ontology was *Name of Country or Region*, was changed to *Name*. The relationship set in the target ontology between *Country* and *Name* is functional, but the corresponding relationship set in the source ontology was not functional. The expert chose to resolve the constraint conflict by making the non-functional relationship set functional. The expert used our tool to resolve these issues and merge the result as Figure 5.3 shows.

It took 65 minutes for an expert using the OntologyEditor functions to integrate 12 mini-ontologies into a domain ontology. On the other hand, it only took 30, 26, and 25 minutes for the expert to map and merge the 12 ontologies using the manual, semi-automatic, and automatic modes of our tool, respectively. For this test case, our tool saved significant time for the user. Using the tool required less than half the time for all modes of operations.

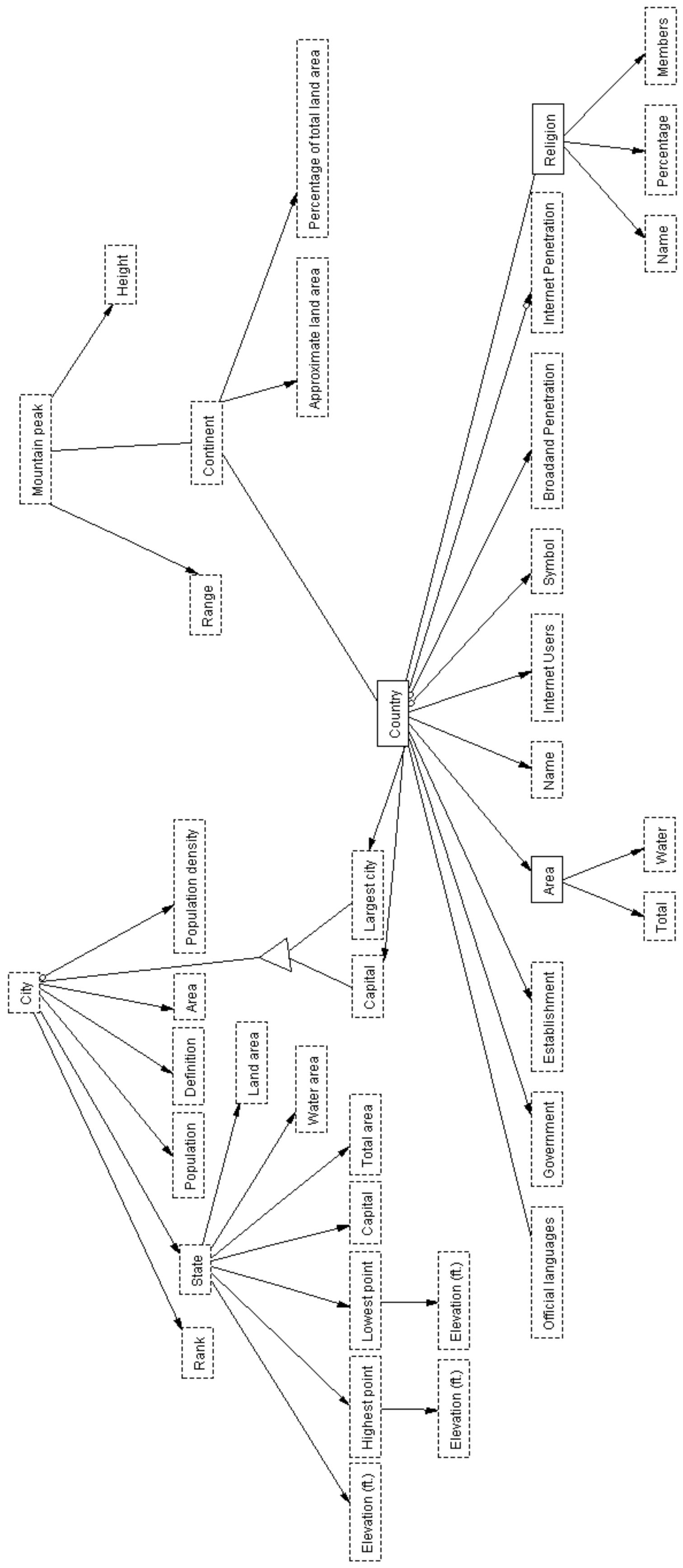


Figure 5.1: The Final Merged Ontology Based on Test Case

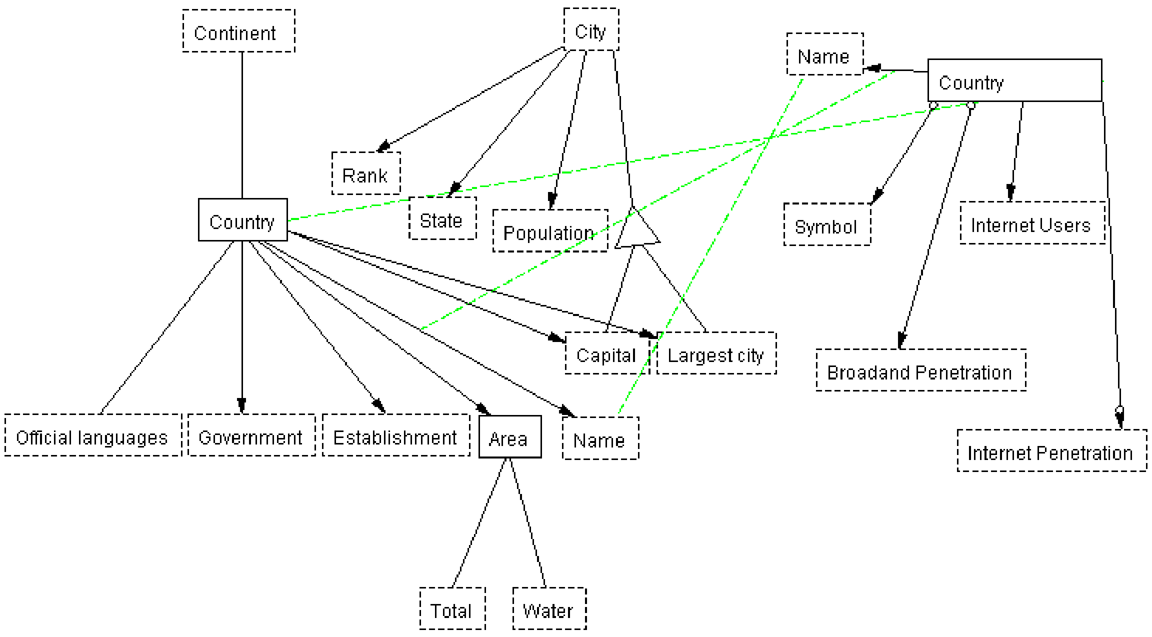


Figure 5.2: Two Mapped Ontologies in Test Case

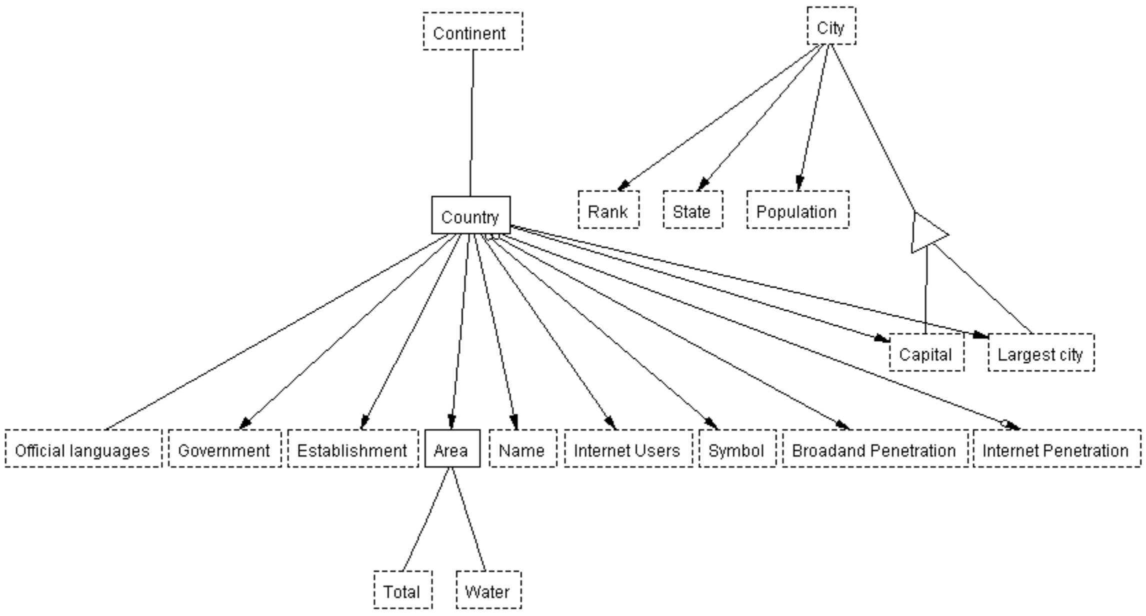


Figure 5.3: A Merged Growing Ontology in Test Case

Chapter 6

Conclusion and Future Work

6.1 Summary

For the thesis, several significant components of the third part of the overall TANGO project were implemented. To prepare for ontology mapping and merging, the OntologyEditor was augmented, so that it could load and simultaneously display two ontologies — the target ontology and the source ontology. The current OntologyEditor was also augmented to support a textual view in addition to a graphical view. For both the graphical view and the textual view, a means was devised and implemented to allow a user to manually map ontology components in the source and target ontology to one another. The tool also provides IDS statements to allow users to resolve conflicts that arise during the mapping process. These were implemented as well, as was a plug-in API for developers to add mapping and merging algorithms to the tool. To show the usage of the API, a naive mapping algorithm and a naive data merge algorithm were also implemented.

6.2 Conclusions

In this thesis, we introduced a tool to help users to preform ontology mapping and merging. The tool has three modes for mapping: manual, semi-automatic, and automatic. We tested the system using 12 mini-ontologies in the geopolitical domain. We compared the time for task completion with the three modes of our tool and without using our tool. The results for the chosen test cases indicated that our system enables better user performance. Although we only tested the system using the geopolitical domain, our tool is designed for all application domains.

6.3 Future Work

As is typical in software projects, implementing an original design reveals opportunities for improvements. With hindsight, we discovered that mapping relationship sets to relationship sets without requiring any prior mapping of related object sets should not only be possible but would likely be preferable. The IDS interactions could be more complex, but in most cases, there would be fewer interactions with the user and a more expeditious specification of mappings. Also, it appears that mapping generalization/specializations and aggregations in a similar way would be more expeditious.

Besides these improvements, we also note that some OSM components were not included in the original design. It was felt that these components would not likely arise in the context of the TANGO project. In particular, co-occurrence constraints for relationship sets was omitted, as was an object/object-set mapping. However, the tool can stand on its own as an ontology mapping and merging tool, and, to be complete, these components should be included.

As a next step toward resolving the ontology merging and mapping problem, we would like to improve our tool by using the API to plug in more sophisticated mapping and merging algorithms. Then we should carry out more comprehensive experimental testing. As it currently stands, our tool is ready for integration with other components of the TANGO project. With enhanced mapping and merging algorithms, our tool should be able to satisfactorily perform its role in automatically or semi-automatically converting collections of related tables into ontologies.

Bibliography

[1] www.w3.org/2001/11/IsaViz/, 2004. 1

[2] ezOWL. iweb.etri.re.kr/ezowl/, 2006. 1

[3] GKB Editor. www.ai.sri.com/gkb/, 2006. 1

[4] TOPKAT - the open practical knowledge acquisition toolkit. www.aiai.ed.ac.uk/jkk/topkat.html, 2006. 2

[5] S. Bechhofer, I. Horrocks, C. Goble, and R. Stevens. OilEd: A reasonable ontology editor for the semantic web. *Lecture Notes in Computer Science*, 2174:396–405, 2001. 1

[6] J. Biskup and D.W. Embley. Extracting information from heterogeneous information sources using ontologically specified target views. *Information Systems*, 28(3):169–212(44), May 2003. 11

[7] P. Cimiano and J. Völker. Text2Onto - a framework for ontology learning and data-driven change discovery. In *Proceedings of the 10th International Conference on Applications of Natural Language to Information Systems (NLDB)*, pages 227–238, Alicante, Spain, June 2005. 1

[8] M. Denny. Ontology tools survey, revisited. www.xml.com/pub/a/2004/07/14/onto.html, 2004. 1

[9] H.-H. Do and E. Rahm. COMA: a system for flexible combination of schema matching approaches. In *Proceedings of the 28th international conference on Very Large Data Bases (VLDB2002)*, pages 610–621, August 2002. 3

[10] D.W. Embley. *Object Database Development: Concepts and Principles.* Addison-Wesley, Reading Massachusetts, 1998. 19

[11] J. Euzenat and P. Shvaiko. *Ontology matching.* Springer-Verlag, Heidelberg (DE), 2007. 3

[12] S. Falconer, N.N., and M.-A. Storey. Ontology mapping - a user survey. In Pavel Shvaiko, Jrme Euzenat, Fausto Giunchiglia, and Bin He, editors, *Proceedings of the Workshop on Ontology Matching (OM2007) at ISWC/ASWC2007*, pages 113–125, Busan, South Korea, November 2007. 3

[13] J.H. Gennari, M.A. Musen, R.W. Fergerson, W.E. Grosso, M. Crubzy, H. Eriksson, N.F. Noy, and S.W. Tu. The evolution of Protege: An environment for knowledge-based systems development. *International Journal of Human Computer Studies*, 58(1):89–123, 2003. 1

[14] M. Touzani P. Valtchev J. Euzenat, D. Loup. Ontology alignment with OLA. In *Proceedings of the 3rd International Workshop on Evaluation of Ontology-based Tools*, pages 60–91, November 2004. 3

[15] S. Lamparter, M. Ehrig, and C. Tempich. Knowledge extraction from classification schemas. In *Proceedings of CoopIS/DOA/ODBASE*, pages 618–636, Agia Napa, Cyprus, 2004. 2

[16] S.W. Liddle, D.W. Embley, and S.N. Woodfield. An active, object-oriented, model-equivalent programming language. pages 333–361, 2000. 19

[17] T. Liebig and O. Noppens. OntoTrack: Fast browsing and easy editing of large ontologies. In *Proceedings of The Second International Workshop on Evaluation of Ontology-based Tools (EON2003)*, Sanibel Island, Florida, October 2003. 1

[18] A. Maedche and S. Staab. Ontology learning. In *Handbook on Ontologies*, pages 173–190. 2004. 1

[19] R.J. Miller, M.A. Hernández, L.M. Haas, L. Yan, C.T. Howard, R.F., and L. Popa. The Clio project: managing heterogeneity. *SIGMOD Record (ACM Special Interest Group on Management of Data)*, 30(1):78–83, 2001. 3

[20] G.A. Modica, A. Gal, and H.M. Jamil. The use of machine-generated ontologies in dynamic information seeking. In *Proceedings of the 9th International Conference on Cooperative Information Systems (CoopIS01)*, pages 433–448, London, England, September 2001. 2

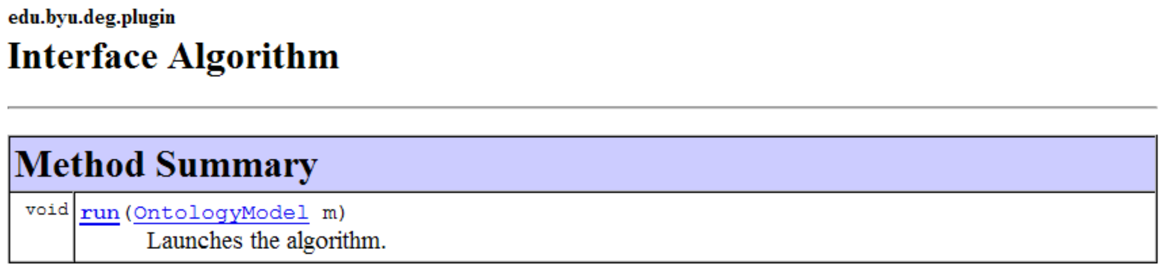
[21] N.F. Noy and M.A. Musen. The PROMPT suite: interactive tools for ontology merging and mapping. *International Journal of Human-Computer Studies*, 59(6):983–1024, 2003. 3

[22] Y. A. Tijerino, D. W. Embley, D. W. Lonsdale, Y. Ding, and G. Nagy. Toward ontology generation from tables. *World Wide Web: Internet and Web Information Systems*, 8(3):251–285, September 2004. 2

[23] A. Wessman, S.W. Liddle, and D.W. Embley. A generalized framework for an ontology-based data-extraction system. In *Proceedings of the 4th International Conference on Information Systems Technology and its Applications (ISTA05))*, pages 239–253, Palmerston North, New Zealand, 2005. 5

Appendix A

API for Mapping and Data Merging Algorithms



edu.byu.deg.plugin

Interface Algorithm

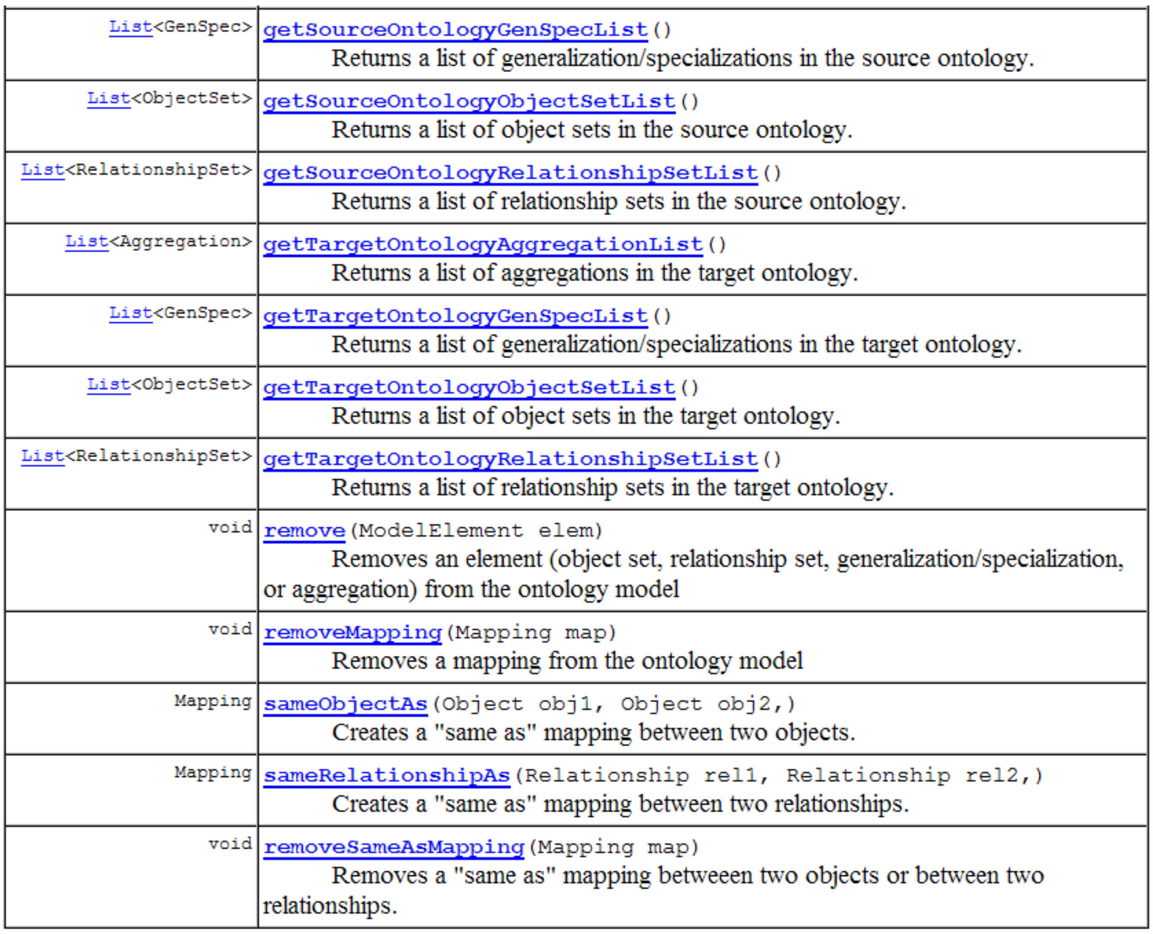
Method Summary	
void	`run(OntologyModel m)` Launches the algorithm.

Figure A.1: The Method to be Implemented for Mapping and Data Merge Algorithms

edu.byu.deg.plugin

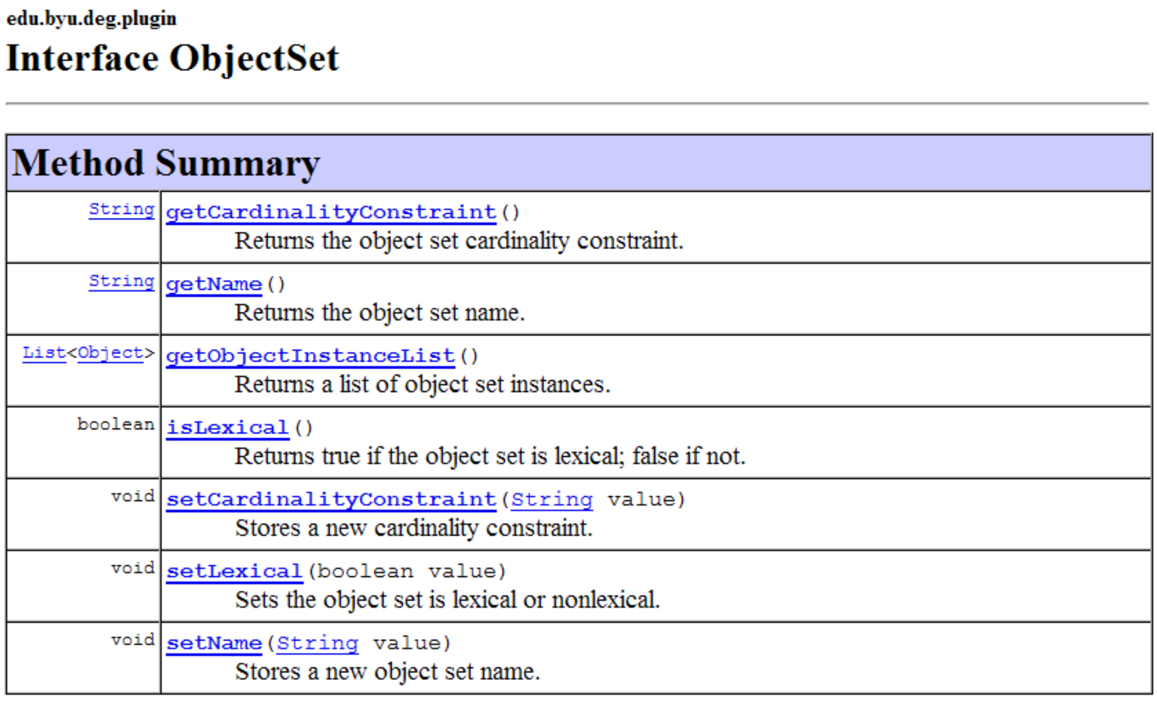
Interface OntologyModel

Method Summary	
`Aggregation`	`createAggregationInSourceOntology(ObjectSet parentObjectSet, List<ChildRelSetConnection> chdCons)` Creates an aggregation in the source ontology.
`Aggregation`	`createAggregationInTargetOntology(ObjectSet parentObjectSet, List<ChildRelSetConnection> chdCons)` Creates an aggregation in the target ontology.
`ChildRelSetConnection`	`createChildRelSetConnection(ObjectSet obj, String parentSideConstraint, String childSideConstraint, boolean parentSideFunctional, boolean childSideFunctional, boolean parentSideOptional, boolean childSideOptional)` Creates a childrelset connection for an aggregation.
`GenSpec`	`createGenSpecInSourceOntology(String constraint, List<ObjectSet> genObjectSetList, List<ObjectSet> specObjectSetList)` Creates a generalization/specialization in the source ontology.
`GenSpec`	`createGenSpecInTargetOntology(String constraint, List<ObjectSet> genObjectSetList, List<ObjectSet> specObjectSetList)` Creates a generalization/specialization in the target ontology.
`Mapping`	`createMapping(ModelElement elem1, ModelElement elem2)` Creates a mapping between object sets, relationship sets, aggregations, or generalization/specialization.
`Object`	`createObject(ObjectSet objRef, String value)` Creates an object which holds a data instance value of an object set.
`ObjectSet`	`createObjectSetInSourceOntology(String name, String cardinalityConstraint, boolean isLexical)` Creates an object set in the source ontology.
`ObjectSet`	`createObjectSetInTargetOntology(String name, String cardinalityConstraint, boolean isLexical)` Creates an object set in the target ontology.
`Relationship`	`createRelationship(RelationshipSet relationshipsetRef, List<Object> objectList)` Creates a relationship which holds a data instance value of a relationship set.
`RelationshipSet`	`createRelationshipSetInSourceOntology(String name, List<RelSetConnection> relCons)` Creates a relationship set in the source ontology.
`RelationshipSet`	`createRelationshipSetInTargetOntology(String name, List<RelSetConnection> relCons)` Creates a relationship set in the target ontology.
`RelSetConnection`	`createRelSetConnection(ObjectSet obj, String participationConstraint, boolean functional, boolean optional)` Creates a connection component of a relationship set (a relset connection).
`List<Aggregation>`	`getSourceOntologyAggregationList()` Returns a list of aggregations in the source ontology.



List<GenSpec>	getSourceOntologyGenSpecList() Returns a list of generalization/specializations in the source ontology.
List<ObjectSet>	getSourceOntologyObjectSetList() Returns a list of object sets in the source ontology.
List<RelationshipSet>	getSourceOntologyRelationshipSetList() Returns a list of relationship sets in the source ontology.
List<Aggregation>	getTargetOntologyAggregationList() Returns a list of aggregations in the target ontology.
List<GenSpec>	getTargetOntologyGenSpecList() Returns a list of generalization/specializations in the target ontology.
List<ObjectSet>	getTargetOntologyObjectSetList() Returns a list of object sets in the target ontology.
List<RelationshipSet>	getTargetOntologyRelationshipSetList() Returns a list of relationship sets in the target ontology.
void	remove(ModelElement elem) Removes an element (object set, relationship set, generalization/specialization, or aggregation) from the ontology model
void	removeMapping(Mapping map) Removes a mapping from the ontology model
Mapping	sameObjectAs(Object obj1, Object obj2,) Creates a "same as" mapping between two objects.
Mapping	sameRelationshipAs(Relationship rel1, Relationship rel2,) Creates a "same as" mapping between two relationships.
void	removeSameAsMapping(Mapping map) Removes a "same as" mapping betweeen two objects or between two relationships.

Figure A.2: The Methods for the Ontology Model

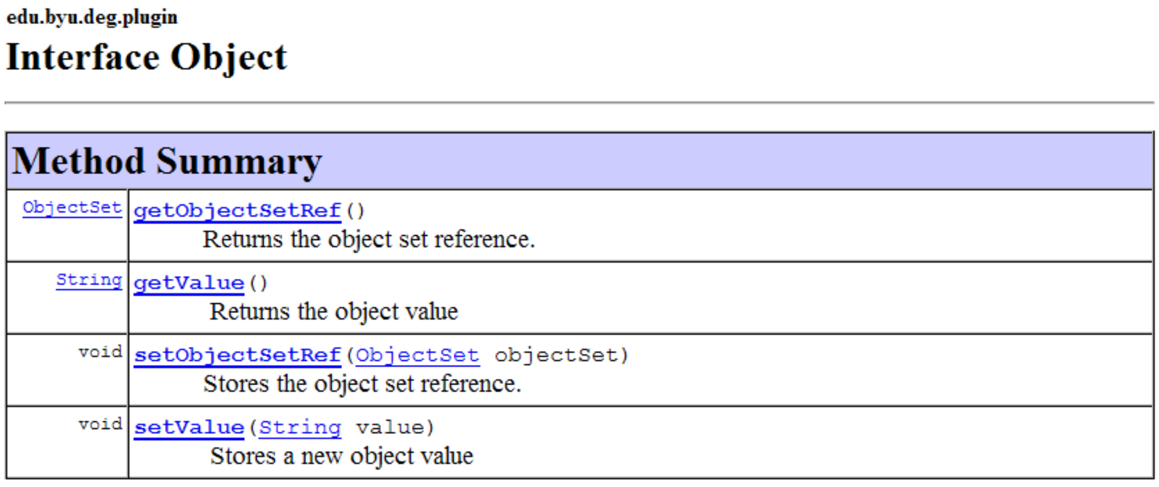


edu.byu.deg.plugin

Interface ObjectSet

Method Summary	
`String`	`getCardinalityConstraint()` Returns the object set cardinality constraint.
`String`	`getName()` Returns the object set name.
`List<Object>`	`getObjectInstanceList()` Returns a list of object set instances.
`boolean`	`isLexical()` Returns true if the object set is lexical; false if not.
`void`	`setCardinalityConstraint(String value)` Stores a new cardinality constraint.
`void`	`setLexical(boolean value)` Sets the object set is lexical or nonlexical.
`void`	`setName(String value)` Stores a new object set name.

Figure A.3: The Methods for ObjectSet

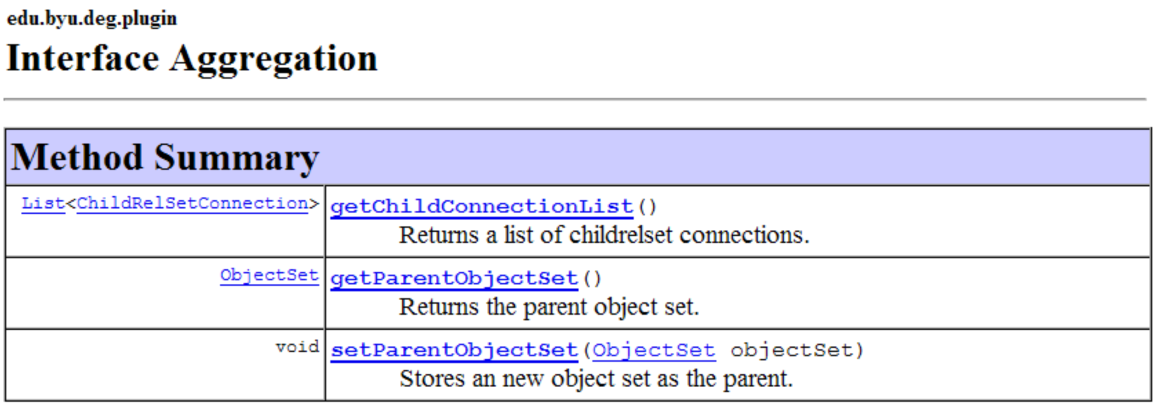


edu.byu.deg.plugin

Interface Object

Method Summary	
`ObjectSet`	`getObjectSetRef()` Returns the object set reference.
`String`	`getValue()` Returns the object value
`void`	`setObjectSetRef(ObjectSet objectSet)` Stores the object set reference.
`void`	`setValue(String value)` Stores a new object value

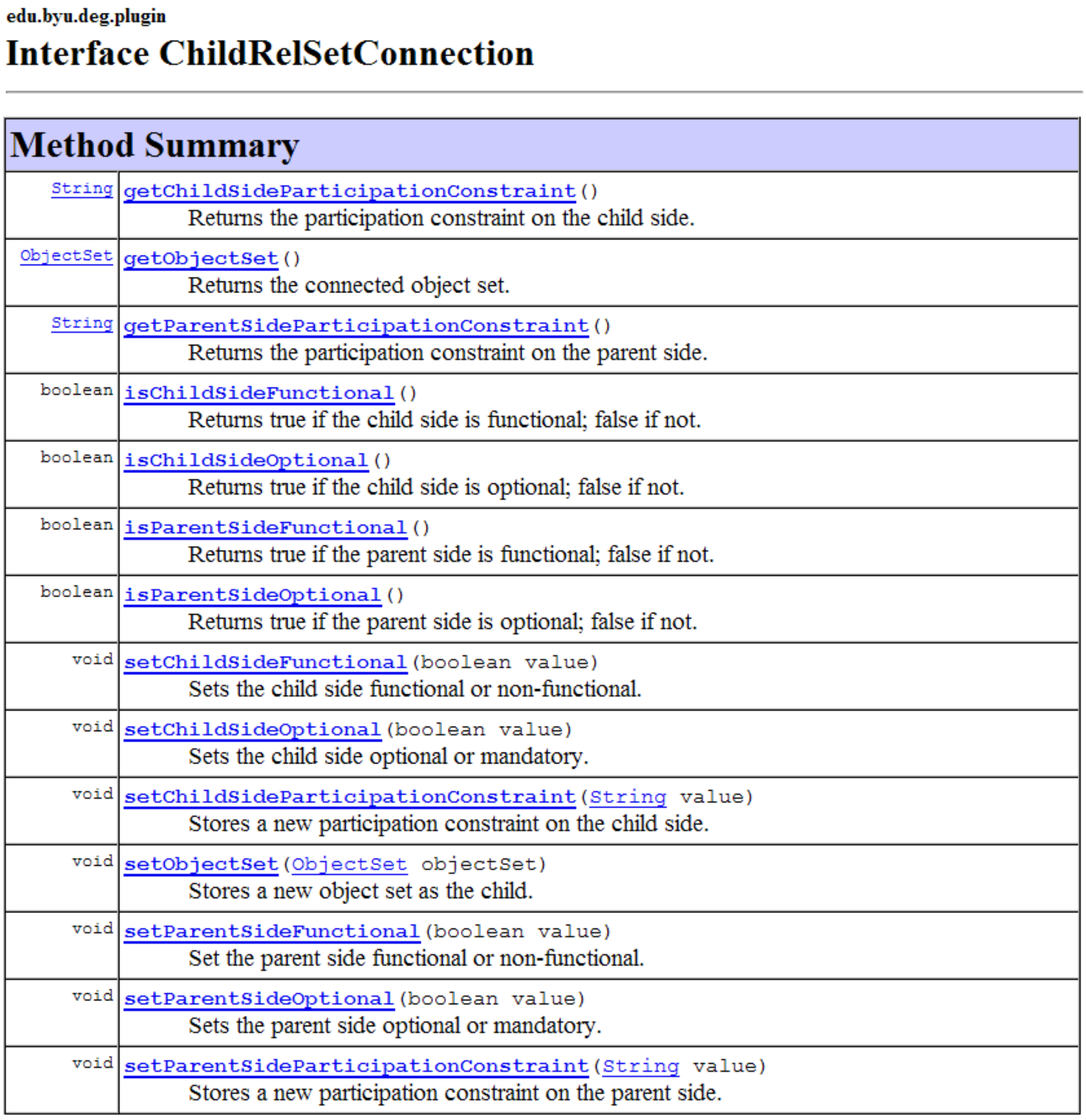
Figure A.4: The Methods for Object


edu.byu.deg.plugin

Interface Aggregation

Method Summary	
`List<ChildRelSetConnection>`	`getChildConnectionList()` Returns a list of childrelset connections.
`ObjectSet`	`getParentObjectSet()` Returns the parent object set.
`void`	`setParentObjectSet(ObjectSet objectSet)` Stores an new object set as the parent.

Figure A.5: The Methods for Aggregation

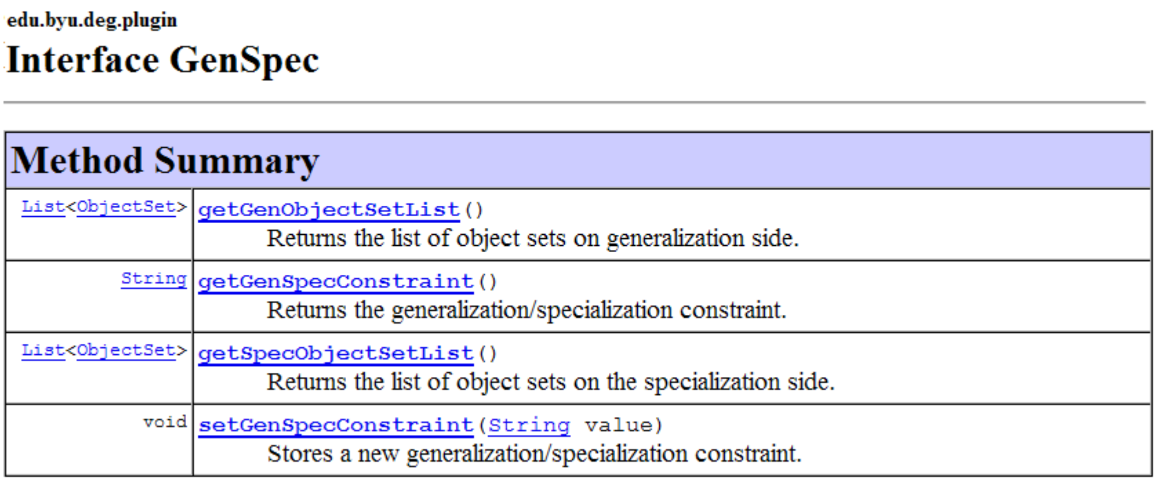


edu.byu.deg.plugin

Interface ChildRelSetConnection

Method Summary	
`String`	`getChildSideParticipationConstraint()` Returns the participation constraint on the child side.
`ObjectSet`	`getObjectSet()` Returns the connected object set.
`String`	`getParentSideParticipationConstraint()` Returns the participation constraint on the parent side.
`boolean`	`isChildSideFunctional()` Returns true if the child side is functional; false if not.
`boolean`	`isChildSideOptional()` Returns true if the child side is optional; false if not.
`boolean`	`isParentSideFunctional()` Returns true if the parent side is functional; false if not.
`boolean`	`isParentSideOptional()` Returns true if the parent side is optional; false if not.
`void`	`setChildSideFunctional(boolean value)` Sets the child side functional or non-functional.
`void`	`setChildSideOptional(boolean value)` Sets the child side optional or mandatory.
`void`	`setChildSideParticipationConstraint(String value)` Stores a new participation constraint on the child side.
`void`	`setObjectSet(ObjectSet objectSet)` Stores a new object set as the child.
`void`	`setParentSideFunctional(boolean value)` Set the parent side functional or non-functional.
`void`	`setParentSideOptional(boolean value)` Sets the parent side optional or mandatory.
`void`	`setParentSideParticipationConstraint(String value)` Stores a new participation constraint on the parent side.

Figure A.6: The Methods for ChildRelSetConnection for Aggregation

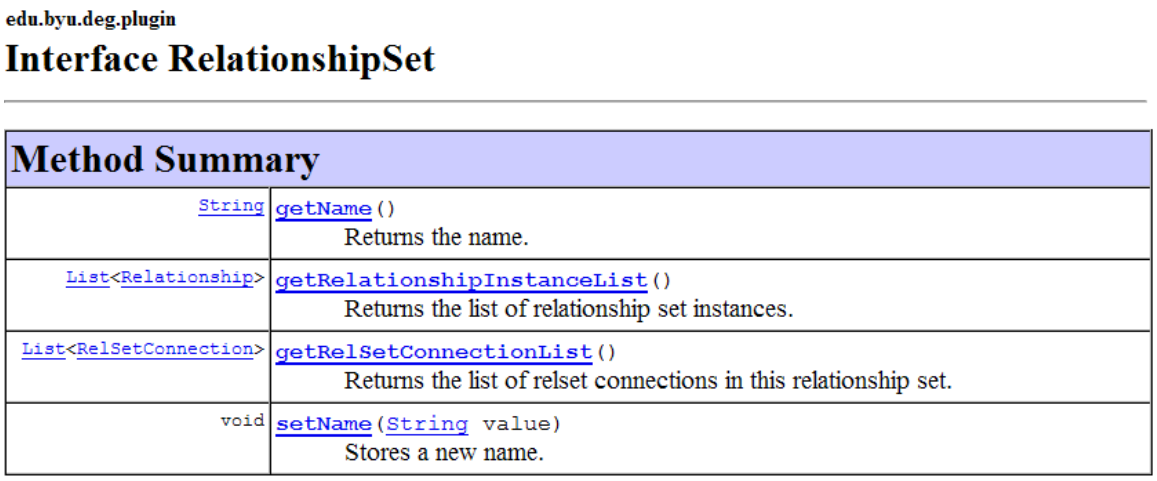


edu.byu.deg.plugin

Interface GenSpec

Method Summary	
List<ObjectSet>	getGenObjectSetList() Returns the list of object sets on generalization side.
String	getGenSpecConstraint() Returns the generalization/specialization constraint.
List<ObjectSet>	getSpecObjectSetList() Returns the list of object sets on the specialization side.
void	setGenSpecConstraint(String value) Stores a new generalization/specialization constraint.

Figure A.7: The Methods for Generalization/Specialization

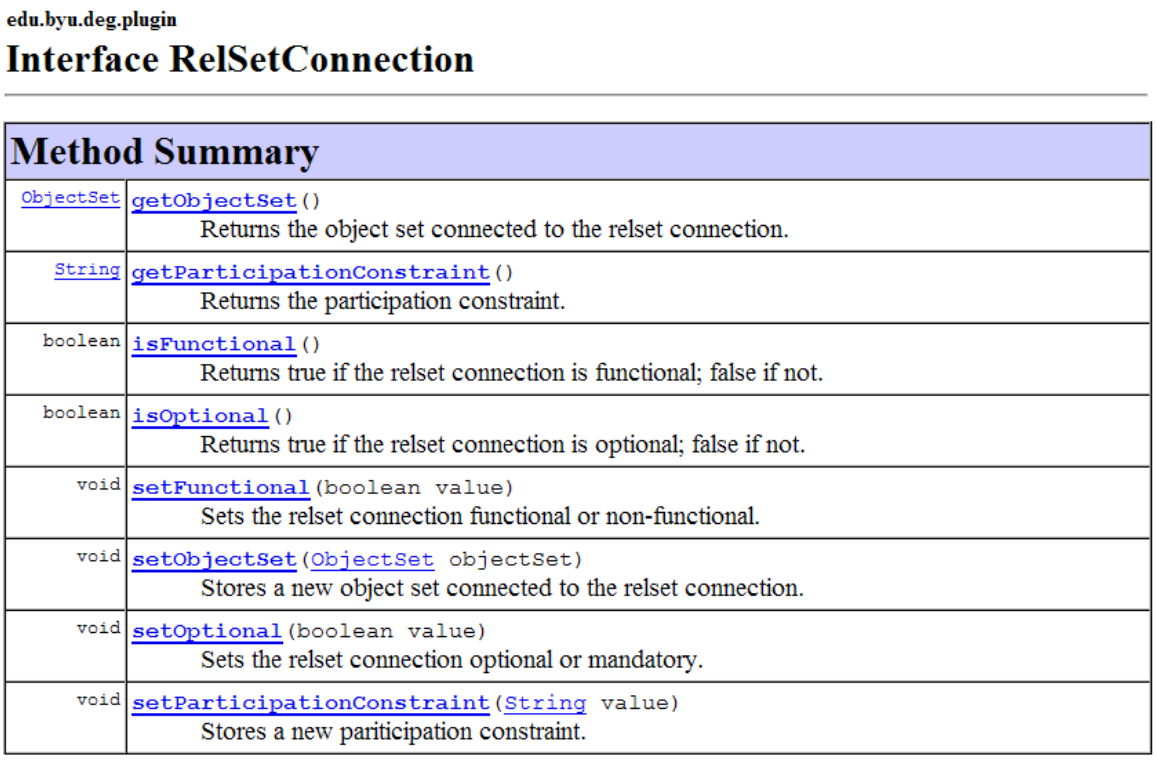


edu.byu.deg.plugin

Interface RelationshipSet

Method Summary	
String	getName() Returns the name.
List<Relationship>	getRelationshipInstanceList() Returns the list of relationship set instances.
List<RelSetConnection>	getRelSetConnectionList() Returns the list of relset connections in this relationship set.
void	setName(String value) Stores a new name.

Figure A.8: The Methods for Relationship Set

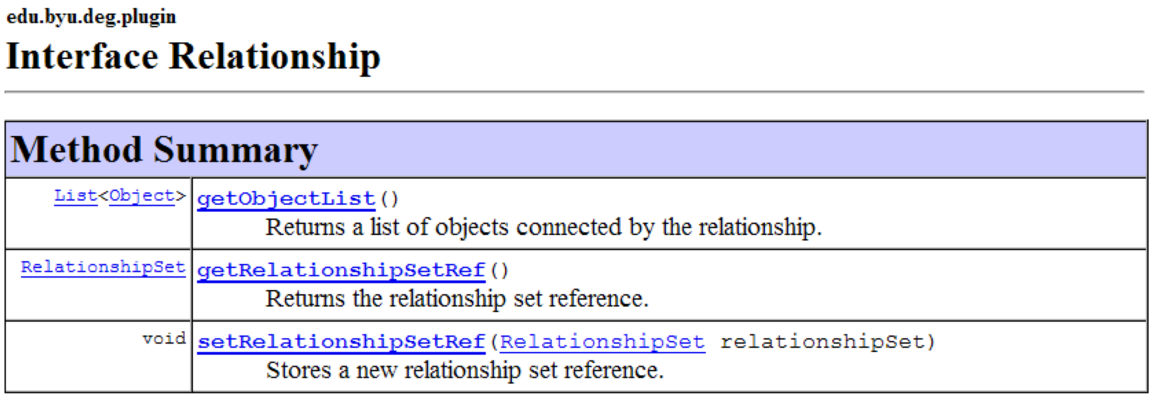


edu.byu.deg.plugin

Interface RelSetConnection

Method Summary	
ObjectSet	getObjectSet() Returns the object set connected to the relset connection.
String	getParticipationConstraint() Returns the participation constraint.
boolean	isFunctional() Returns true if the relset connection is functional; false if not.
boolean	isOptional() Returns true if the relset connection is optional; false if not.
void	setFunctional(boolean value) Sets the relset connection functional or non-functional.
void	setObjectSet(ObjectSet objectSet) Stores a new object set connected to the relset connection.
void	setOptional(boolean value) Sets the relset connection optional or mandatory.
void	setParticipationConstraint(String value) Stores a new pariticipation constraint.

Figure A.9: The Methods for RelSetConnection for Relationship Set



edu.byu.deg.plugin

Interface Relationship

Method Summary	
List<Object>	getObjectList() Returns a list of objects connected by the relationship.
RelationshipSet	getRelationshipSetRef() Returns the relationship set reference.
void	setRelationshipSetRef(RelationshipSet relationshipSet) Stores a new relationship set reference.

Figure A.10: The Methods for Relationship

Appendix B

Test Cases

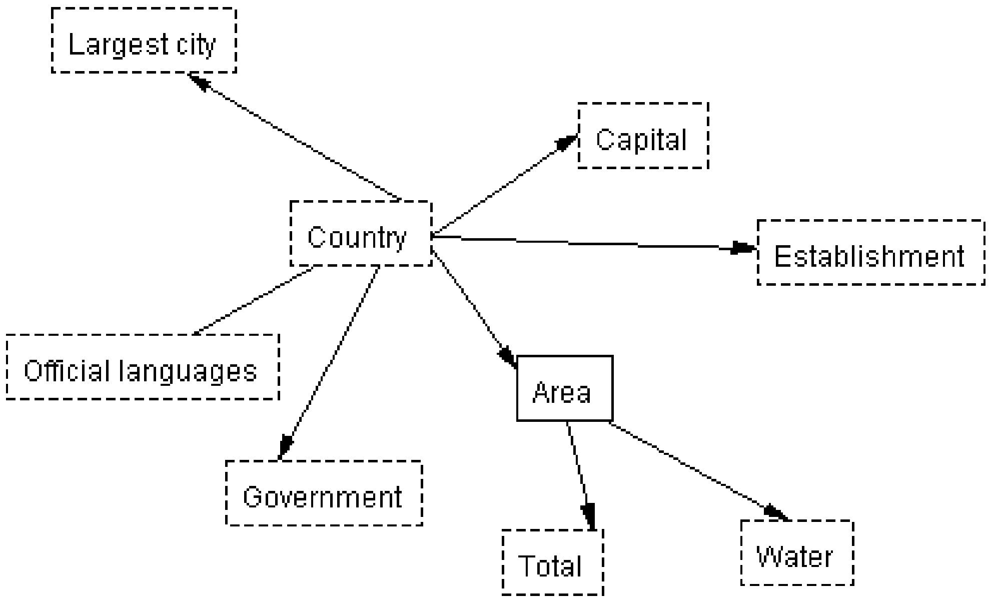


Figure B.1: Test Mini-Ontology 1

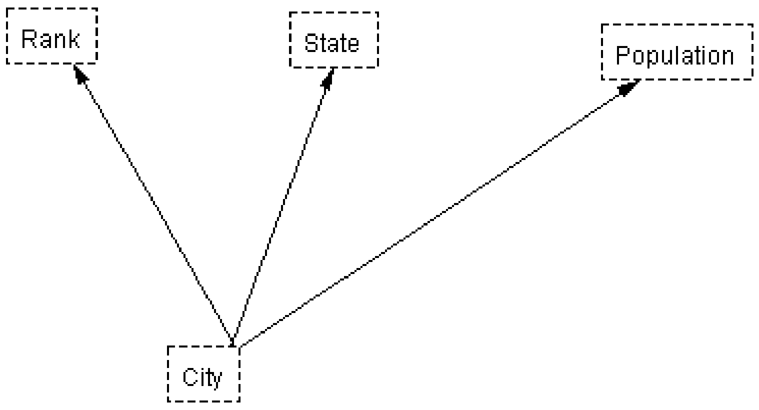


Figure B.2: Test Mini-Ontology 2

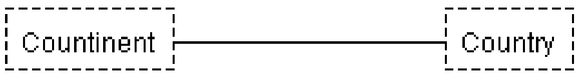


Figure B.3: Test Mini-Ontology 3

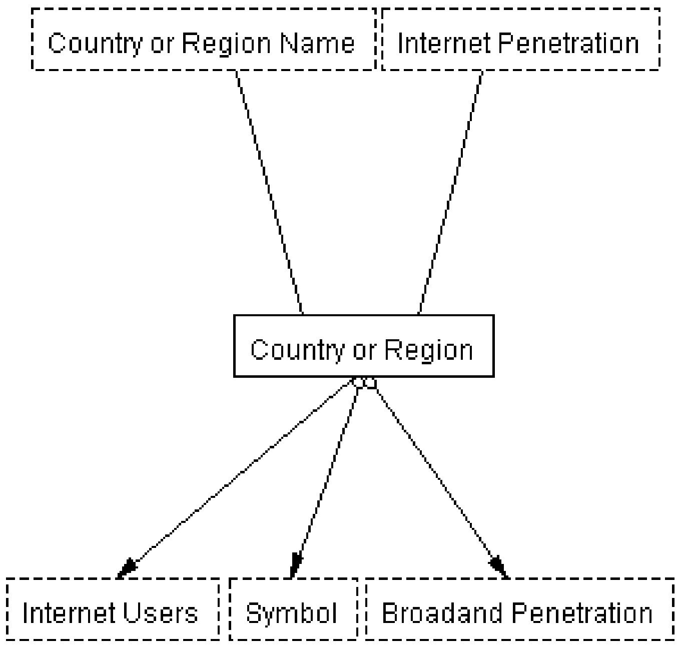


Figure B.4: Test Mini-Ontology 4

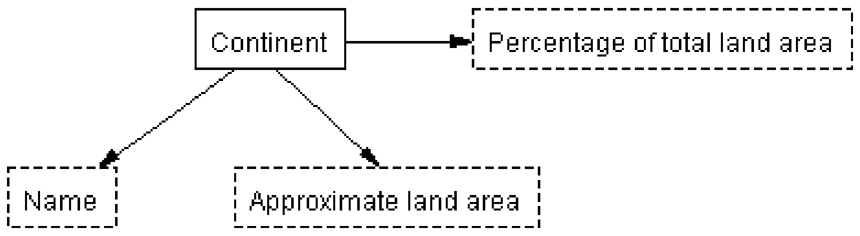


Figure B.5: Test Mini-Ontology 5

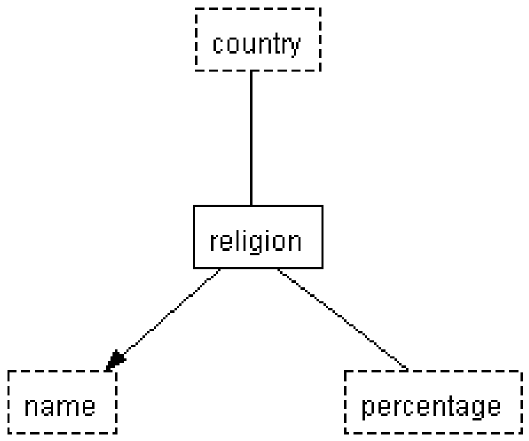


Figure B.6: Test Mini-Ontology 6

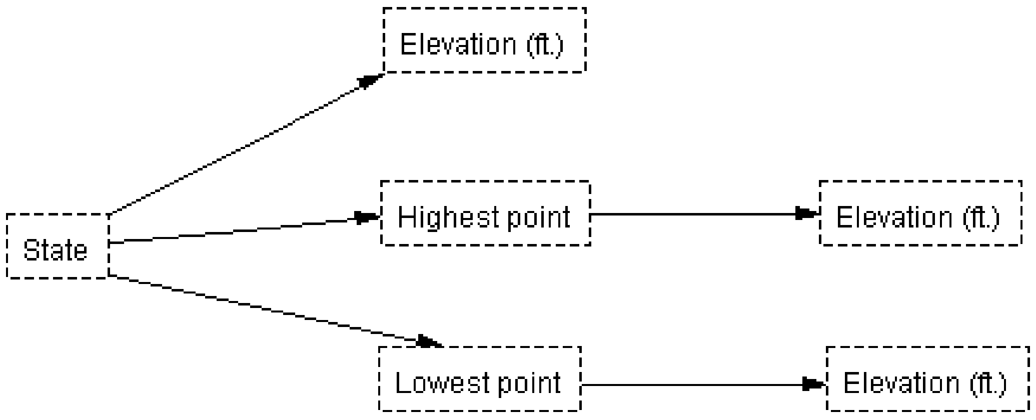


Figure B.7: Test Mini-Ontology 7

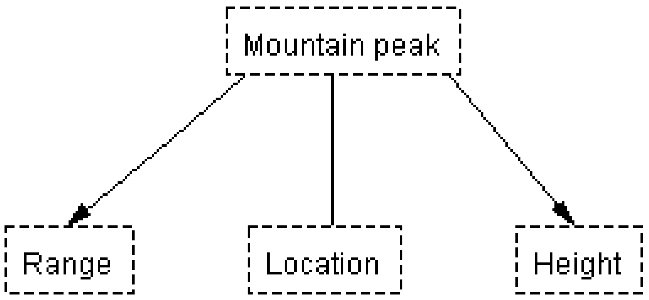


Figure B.8: Test Mini-Ontology 8

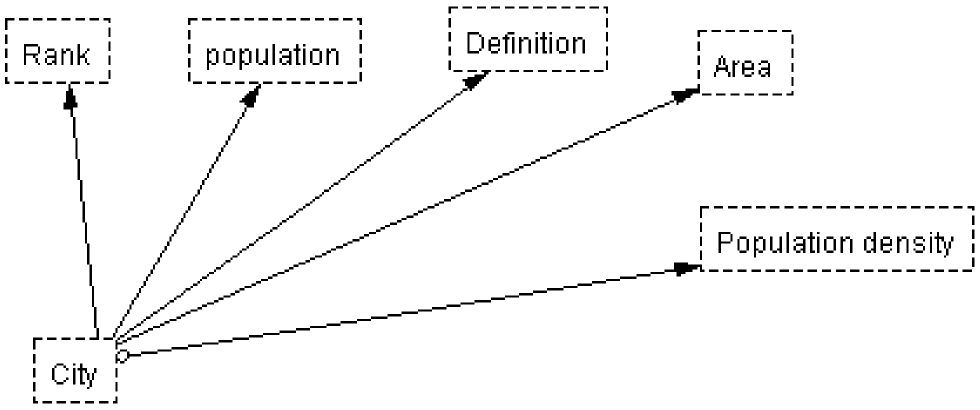


Figure B.9: Test Mini-Ontology 9

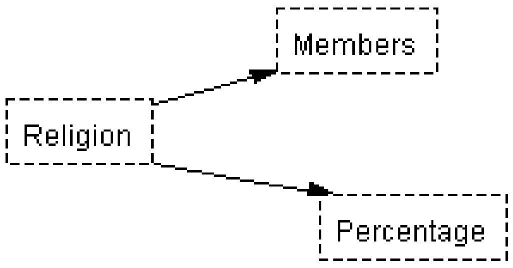


Figure B.10: Test Mini-Ontology 10

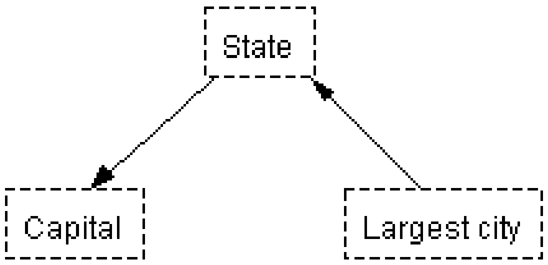


Figure B.11: Test Mini-Ontology 11

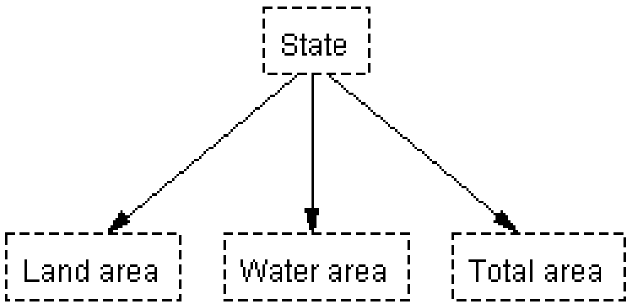


Figure B.12: Test Mini-Ontology 12