Dependency Analysis in Ontology-driven Content-based Systems

Yalemisew M. Abgaz[1], Muhammad Javed[2], Claus Pahl[3]

Centre for Next Generation Localization (CNGL),
School of Computing, Dublin City University, Dublin 9, Ireland
{yabgaz[1]|mjaved[2]|cpahl[3]}@computing.dcu.ie

Abstract. Ontology-driven content-based systems are content-based systems (ODCBS) that are built to provide a better access to information by semantically annotating the content using ontologies. Such systems contain ontology layer, annotation layer and content layer. These layers contain semantically interrelated and interdependent entities. Thus, a change in one layer causes many unseen and undesired changes and impacts that propagate to other entities. Before any change is implemented in the ODCBS, it is crucial to understand the impacts of the change on other ODCBS entities. However, without getting these dependent entities, to which the change propagates, it is difficult to understand and analyze the impacts of the requested changes. In this paper we formally identify and define relevant dependencies, formalizing them and present a dependency analysis algorithm. The output of the dependency analysis serves as an essential input for change impact analysis process that ensures the desired evolution of the ODCBS.
Keywords: Dependency analysis, Change impact analysis, Content-based systems, Ontology-driven content-based systems.

1 Introduction

Ontology-driven content-based systems are content-based information systems that are built to provide a better access to information for both humans and machines by semantically enriching the content using ontologies. In such systems, using semantic annotation, the ontologies provide rich semantics to the content at hand [1][2][3]. To achieve this purpose, we proposed a layered framework [4] which contains the ontology, the annotation and the content layers. A continual change of entities in the layers causes the ODCBS to evolve dynamically [5].

Changes in ODCBS are complex as a result of the interdependence of the entities at different layers, the nature of the changes and the semantics involved in ODCBS. When an entity changes, the change propagates to other entities resulting intermediate changes to other dependent entities [6]. The propagation is towards the dependent entities of the changing entity. Because the interdependence between entities in the layers involves semantics, identifying the dependent entities is an arduous and complex and time consuming task. It is aggravated by the evolution strategies [7] which require further analysis on the nature of the dependencies within and across the layers [4].

Understanding these dependencies and their nature is crucial for analyzing the impacts of changes in the ODCBS. A systematic and careful analysis for identifying dependent entities and analyzing the propagation of impacts to dependent entities is of vital importance in the evolution process [7][6]. Some key features of our approach are:

- providing the theoretical foundation for dependency analysis in ontology-driven content-bases systems.
- identifying the crucial and relevant dependencies that exist within and among the layers of the ODCBS. These dependencies are used to generate change operations [8] and to analyze impacts of change operations in ODCBS.
- providing formal definition of the identified relevant dependencies.
- providing algorithms to identify dependent entities for further analysis.

This research benefits us in different ways. It will serve as a vital input for generating change operations for different evolution strategies. It also serves as an input for change impact analysis process. It facilitates the visibility of the affected entities, improves the integrity of the ODCBS and makes the evolution process smooth and predictable.

This paper is organized as follows: Section 2 gives an overview of ODCBSs, its layered architecture and its graph based representation. In section 3, we present dependencies in ODCBS, their types and selected algorithms to identify dependent entities. Section 4 focuses on evaluation using empirical studies. Related work is given in section 5 and conclusion and future work in section 6.

2 Overview of Ontology-Driven Content-Based Systems

We represent an ODCBS using graph-based formalism. Graphs are selected for their known efficiency and similarity to ontology taxonomy. A full discussion of the ODCBS architecture is found in [4].

An ODCBS is represented as graph $G = G_o \cup G_a \cup Cont$ where: G_o is the ontology graph, G_a is the annotation graph and $Cont$ is the content set.

An ***ontology*** O is represented by a direct labelled graph $G_o = (N_o, E_o)$ where: $N_o = \{n_{o1}, n_{o2}, \ldots, n_{om}\}$ is a finite set of labelled nodes that represent classes, data properties, object properties etc. $E_o = \{e_{o1}, e_{o2} \ldots, e_{om}\}$ is a finite set of labelled edges and $e_{oi} = (n_1, \alpha, n_2)$ where: n_1 and n_2 are members of N_o and the label of an edge represented by α = {subclassOf, intersectionOf, minCardinality, maxCardinality...}. The labels may indicate the relationship (dependency) between the nodes. A ***content*** represented by $Cont$ can be viewed as a set of documents $D = \{d_1, d_2, d_3 .... d_n\}$ where: d_i represents a single document or part of a document which can be mapped to nodes in the annotation graph. An ***annotation*** $Anot$ is represented by a direct labelled graph $G_a = (N_a, E_a)$ where: N_a and E_a are finite set of labelled nodes and edges respectively. An edge $E_a = (n_{a1}, \alpha_a, n_{a2})$ where $n_{a1} \in \{Cont\}$ as a subject, $n_{a2} \in \{Cont\} \cup \{O\}$ as an object and $\alpha_a \in \{O\}$ as a predicate. The graph-based representation of an ODCBS is presented in (Fig. 1)and serves as a running example.

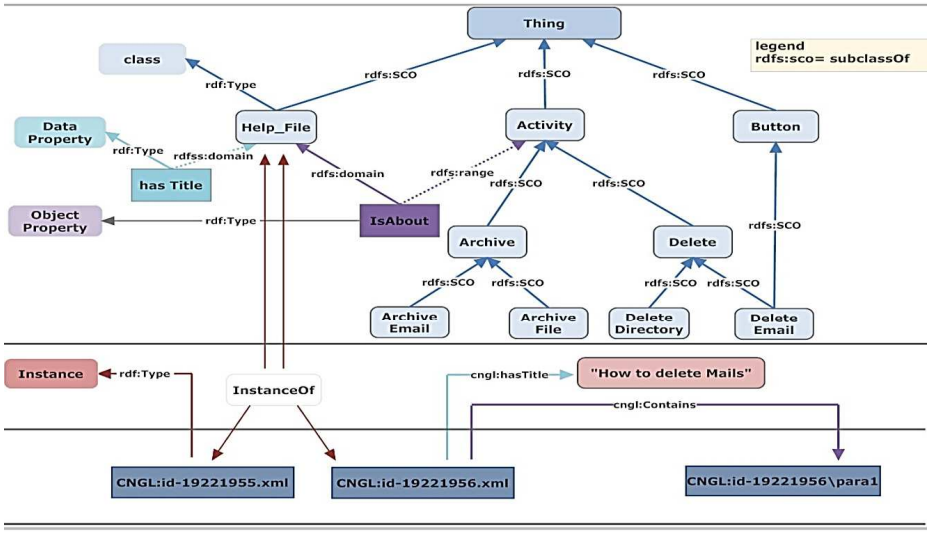

Fig. 1. Graph-based representation of sample ODCBS layered architecture

The type of a node is given by a function $type(n)$ that maps the node to its type (class, instance, data property, object property...). The label of any edge $e = (n_1, \alpha, n_2)$, which is α , is a string given by a function $label(e)$. All the edges of a node n are given by a function $edges(n)$. It returns all the edges as (n, α, m) where n is the target node and m is any node linked to n via α.

3 Dependency in ODCBSs

Characterization, representation and analysis of dependencies within and among the ontology, the annotation and the content layers is subtle and crucial aspect to perform impact analysis [9]. Using an empirical study [10] we discovered different types of dependencies that exist between entities within and among the layers.

Dependency is defined as a relationship between entities where the entities are related to each other by a given relation. Given a dependency between two entities (A and B) in the ODCBS, represented as $Dep(A, B)$, A is the dependent entity and B is the antecedent entity and there is a relationship that relates A to B. Dependency can be unidirectional or bidirectional. ***Dependency Analysis*** is a process of identifying the dependent entities of s given entity.

3.1 Dependency within Layers

In this section, we present the dependencies we identified in each layers of the ODCBS. The following list includes only frequently observed and useful dependencies and is not an exhaustive list.

1. **Concept-Concept Dependency:** Given two class nodes c_i and $c_j \in G_o, c_i$ is dependent on c_j represented by $dep(c_i, c_j)$, if there exist an edge $e_i =$

$(n_1, \alpha, n_2) \in G_o$ such that $(n_1 = c_i) \wedge (n_2 = c_j) \wedge (label(e_i) =$ "$subClassOf$") $\wedge$ $(type(n_1) = type(n_2) =$ "$class$"). Concept-concept dependency is transitive. For example, there is a concept-concept dependency between *Activity* and *Archive*. *Archive* depends on *Activity* because it is a subClass Of *Activity*.

2. **Concept-Axiom Dependency:** Given an axiom edge a_1 and a concept node $c_1 \in G_o$, a_1 is dependent on c_1 represented by $dep(a_1, c_1)$, if there exist an edge $e_i = (n_1, \alpha, n_2) \in G_o$ such that $(n_1 = c_1) \vee (n_2 = c_1) \wedge type(n_1) = type(n_2) =$ "$class$". For example, if we take the concept *Activity* there are three dependent *subClassOf* edges and one dependent *rdfs:range* edge.
3. **Concept-Restriction Dependency :** Given a restriction r_1 and a concept node $c_1 \in G_o$, r_1 is dependent on c_1 represented by $dep(r_1, c_1)$ if there exist an edge $e_i = (n_1, \alpha, n_2) = r_1 \in G_o$ such that $(n_2 = c_1) \wedge type(n_2) =$ "$class$". For example, if we have a restriction (*isAbout, allValuesFrom, Activity*), this specific restriction is dependent on the concept *Activity*.
4. **Property-Property Dependency:** Given two property nodes $p_1, p_2 \in G_o, p_1$ is dependent on p_2 represented by $dep(p_1, p_2)$ if there exist an edge $e_i = (n_1, \alpha, n_2) \in G_o$ such that $(n_1 = p_1) \wedge (n_2 = p_2) \wedge (label(e_i) =$ "$subPropertyOf$") $\wedge type(n_1) = type(n2) =$ "$property$". Here property refers to both data property and object property.
5. **Property-Axiom Dependency:** Given an axiom edge a_1 and a property node $p_1 \in G_o, a_1$ is dependent on p_1 represented by $dep(a_1, p_1)$ if there exist an edge $e_i = (n_1, \alpha, n_2) = a_1 \in G_o$ such that $(n_1 = p_1) \wedge type(n_1) = type(n_2) =$ "$property$".
6. **Property-Restriction Dependency:** Given a restriction edge r_1 and a property node $p_1 \in G_o, r_1$ is dependent on p_1 represented by $dep(r_1, p_1)$ if there exist an edge $e_i = (n_1, \alpha, n_2) \in G_o$ such that $(n_1 = p_1) \vee (n_2 = p_2) \wedge type(n_1) = type(n_2) =$ "$property$".
7. **Axiom-Concept Dependency:** Given an axiom edge a_1 and a concept node $c_1 \in G_o, c_1$ is dependent on a_1 represented by $dep(c_1, a_1)$ if there exist an edge $e_i = (n_1, \alpha, n_2) \in G_o$ such that $(n_1 = c_1) \wedge (label(e_i) =$ "$subClassOf$") $\wedge$ $(type(n_1) =$ "$class$").

3.2 Dependency across Layers

We also observed entities in one layer depending on entities in another layer. These dependencies are treated separately and are discussed below.

Content-annotation dependency. An annotation a_i in the annotation layer is dependent on d_i in the content layer, represented by $dep(a_i, d_i)$, if there exist an edge $e_a = \{n_{ai}, \alpha_a, n_{aj}\} \in G_a$ such that $(n_{ai} = d_i) \vee (n_{aj} = d_i)$. This means a_i is dependent on document d_i if the document is used as a subject or an object of the annotation triple.

Ontology-annotation dependency. The relevant dependencies between entities in the annotation and the ontology layer are presented below.

1. **Concept-Instance Dependency:** Given an instance node i_1 and a concept node $C_1 \in G, i_1$ is dependent on C_1 represented by $dep(i_1, C_1)$ if there exist

an edge $e_i = (n_1, \alpha, n_2) \in G$ such that $(n_1 = i_1) \wedge (n_2 = C_1) \wedge (label(e_i) =$ "$InstanceOf$") $\wedge type(n_1) =$ "$Instance$" $\wedge type(n_2) =$ "$class$". For example, the instance $CNGL : id19221955.xml$ is dependent on the concept $Help\_file$ due to $(CNGL : id - 19221956.xml, instanceOf, Help\_file)$.

2. **Property-Instance property Dependency:** Given an instance property node ip_1 and a property node $p_1 \in G$, ip_1 is dependent on p_1 represented by $dep(ip_1, p_1)$ if there exist an edge $e_i = (n_1, \alpha, n_2) \in G$ such that $(label(e_i) = p) \wedge type(n_1) =$ "$instance$" $\vee type(n_2) =$ "$instance$". For example, in $(CNGL :$ $id19221956.xml$, *cngl:hasTitle*, *How to delete Mails*) the instance property *cngl:hasTitle* is dependent on the property *hasTitle* in the ontology layer.
3. **Axiom-Instance Dependency:** Given an instance node i_1 and an axiom edge $a_1 \in G, i_1$ is dependent on a_1 represented by $dep(i_1, a_1)$ if there exist an edge $e_i = (n_1, \alpha, n_2) \in G$ such that $(n_1 = i_1) \wedge (label(e_i) =$ "$instanceOf$") $\wedge$ $(type(i_1) =$ "$Instance$").

3.3 Types of Dependencies and Dependency Determination

Direct Dependency/Indirect Dependency. Direct dependency is the dependency that exist between two adjacent nodes(n_1, n_2). This means, there is an edge $e_i = (n_1, \alpha, n_2)$. Indirect dependency is a dependency of a node on another by a transitive or intermediate relationship. There exist a set of intermediate edges $(n_1, \alpha, n_x)(n_x, \alpha, n_y)...(n_z, \alpha, n_2)$ that link the two nodes.

Algorithm 1 getDirectDependentClasses(G,c)

```
1: Input: Graph G, Class node c
2: Output: direct dependent classes=d
3: d ← ∅
4: if the node c exists in G then
5:    for each edge e_i = (m, α, c) directed to c do
6:       if label(e_i) = "subClassOf" ∧ type(m) = "class" then
7:          add m to d
8:       end if
9:    end for
10: end if
11: return d
```

A node is considered as dependent node only when it satisfies one or more of the dependencies defined in section 3.1 and 3.2. Algorithm 1 identifies direct dependent entities and it focuses only on class nodes. However, it is implemented for all node types. To get both direct and transitive dependent entities, we expand algorithm 1 to include the transitive dependent entities. Algorithm 2 identifies all direct and transitive dependencies.

Total Dependency/Partial Dependency. Total dependency refers to a dependency when a target node depends only on a single node (articulation

node). That means, there is no other entity that gives meaning to the target entity except the antecedent. Algorithm 3 returns all total dependent classes.

Given two nodes $n_1, n_2 \in G$, n_1 is totally dependent on n_2 represented by $Tdep(n_1, n_2)$ if and only if, $\exists(dep(n_1, n_2)) \wedge \nexists(dep(n_1, n_3))$, where $(n_2 \neq n_3)$. Partial dependency refers to a dependency when the existence of a node depends on more than one node. Two nodes n_1 and $n_2 \in G$, are partially dependent represented by $Pdep(n_1, n_2)$ if and only if, $\exists dep(n_1, n_2) \wedge \exists dep(n_1, n_3)$, where $(n_2 \neq n_3)$. We can reuse algorithm 3 to return the partial dependent classes.

Algorithm 2 getAllDependentClasses(G,c)

```
1: Input : Graph G, Class node c
2: Output: all dependent classes=d
3: d ← ∅
4: Queue Q
5: if the node c exists in G then
6:    DirectDep ← getDirectDependentClasses(G,c)
7:    for each concept c_i in DirectDep do
8:       Q.push(c_i)
9:       if c_i not in d then
10:         add c_i to d
11:      end if
12:   end for
13:   while Q is not empty do
14:      Temp = Q.peek()
15:      getAllDependentClasses(G,Temp)
16:      Q.remove()
17:   end while
18: end if
19: return d
```

Total and partial dependency plays a major role in determining the impacts of a change operation. If a class is totally dependent on the changing class, that class is affected by the change. It becomes orphan concept. But, if that class is partially dependent, the deletion of the class causes only semantic impact. The change makes the partial class neither orphan nor cyclic.

Direct total and partial dependent entities. Direct total dependent entities are entities that are the result of the intersection between total dependent and direct dependent entities. The intersection of the results of algorithm 1 and algorithm 3 gives us the direct total dependent entities. Direct partial dependent entities are entities that are both directly dependent but which are partially dependent entities. These entities play a major role in the impact analysis process.

Limitation of the algorithm. The limitation of the dependency analysis algorithm is related to complex class expressions. The algorithm that separates the partial and total dependencies in such expressions is not fully covered and the algorithm only identifies such expressions as total dependent expressions.

Algorithm 3 getTotalDependentClass(G,c)

```
1: Input : Graph G, Class node c
2: Output: all total dependent classes=d
3: d ← ∅, contained=true
4: Set depCls=∅ ,totalDepCls=∅ ,partialDepCls=∅, super=∅
5: depCls← getAllDependentClasses(G,c)
6: for each concept c_i in depCls do
7:    if  count(getSuperClasses(c_i)=1 then
8:       super ← getSuperClasses(G, c_i)
9:       if  super not in partialDepcls then
10:         add c_i to totalDepCls
11:      end if
12:   else
13:      super ← getSuperClasses(G, c_i)
14:      contained=true
15:      for each sc in super do
16:         if sc not in depCls then
17:            contained=false
18:         end if
19:      end for
20:   end if
21:   if contained=true then
22:      add c_i to totalDepCls
23:   else
24:      add c_i to partialDepCls
25:   end if
26: end for
27: return totalDepCls
```

4 Evaluation

We used the empirical study [10] to evaluate the completeness, accuracy, the adequacy, the transferability and the practical applicability of the solution. The evaluation uses a content-based system built for software help management system to semantically enrich software help files. We used frequent change operations which are used to evolve the ODCBS. For each change, we conducted dependency analysis manually and using the proposed method separately. A comparative result of the dependency analysis conducted for one selected change operation, *delete concept (Activity)*, is presented in table 1. The operation deletes a concept *"Activity"*, but the change propagates to other dependent entities.

The result shows that the proposed method is accurate in that it identified all the dependent entities identified by the manual method. It further identifies entities that are not identified by the manual analysis too. This is mainly attributed to dependent axioms that the manual analysis overlooked or failed to recognize. A similar result is observed in the other selected change operations however, due to space constraint we do not present all of them here.

The solution applied in other domain (university administration) shows a fairly similar result to the results in table 1. It identifies all the dependent entities

that are manually identified and more axioms than the manual method. This shows that the dependency analysis is adequate and transferable to other similar domains. The algorithm gives us a complete list of all the dependent entities

Table 1. Comparison of the manual and automatic method

Entity	**Automatic**			**Manual**
	Total Dependent	Direct Dependent	All Dependent	All Dependent
Concepts	5	2	6	6
Axioms	14	5	14	9
Instances	1	2	2	2
Properties	0	0	0	0

that are identified manually. This makes it complete and guarantees to return all dependent entities. The algorithm can be customized to find dependency to a certain level of depth, which makes it suitable for n-level cascading which seeks dependent entities of a given entity within n node distance.

We can use these entities to analyze change propagation, and to identify impacts of a change operation in different evolution strategies. This allows the users to see which entities are affected, how and why they are affected.

5 Related Work

A closely related work is given by [11]. They conducted a study on validating data instances against ontology evolution to evaluate the validity of data instances. In their research they identify 5 dependencies and two independencies to detect implicit semantic changes and generating semantic views. Our work focuses on dependency analysis to identify all affected entities using the current version and the change operations before they are permanently implemented in the ODCBS.

A related work [12] from a software domain conducted analysis and visualization of behavioural dependencies in UML model. They defined structural and behavioural dependencies, direct and transitive dependences to analyze how one entity depends on another. Their work focuses on identifying and measuring dependencies in UML models. An interesting work done by [13] focuses on dependency analysis using conceptual graphs. Even if the work is more conceptual, it has interesting similarity to our work. They identified dependent and antecedent entities, and further identify impact as an attribute of dependency.

6 Conclusion and Future Work

In this work, we identified relevant dependencies within and across ODCBS layers. We formalized each of the dependencies using graph based formalization. We further developed algorithm that identifies these dependencies which will be used as an input for other phases of ODCBS evolution. The proposed method identifies the relevant dependent entities and the nature of the dependency.

This work is one phase of the bigger change impact analysis research we are conducting for ODCBS systems. The output of this phase will be used as an input

for change operation generation, and further for change impact analysis process. It will be used for analysing optimal implementation of change operations, and change operation orchestration. Our future work will be applying the results of the dependency analysis process for change impact analysis.

Acknowledgment. This material is based upon works supported by the Science Foundation Ireland under Grant No. 07/CE/I1142 as part of the Centre for Next Generation Localisation (www.cngl.ie) at Dublin City University (DCU).

References

1. Gruber, T.R.: A translation approach to portable ontology specifications. Knowledge Acquisition **5**(2) (1993) 199–220
2. Reeve, L., Han, H.: Survey of semantic annotation platforms. In: SAC '05: Proceedings of the 2005 ACM symposium on Applied computing. (2005) 1634–1638
3. Uren, V., Cimiano, P., Iria, J., Handschuh, S., Vargas-Vera, M., Motta, E., Ciravegna, F.: Semantic annotation for knowledge management:requirements and survey of the state of the art. Web Semantics: Science, Services and Agents on World Wide Web. **4**(1) (2006) 14–28
4. Abgaz, Y., Javed, M., Pahl, C.: A framework for change impact analysis of ontology-driven content-based systems. In: On the Move to Meaningful Internet Systems: OTM 2011 Workshops. Lecture Notes in Computer Science. (2011)
5. Gruhn, V., Pahl, C., Wever, M.: Data model evolution as basis of business process management. In: Proceedings of the 14th International Conference on Object-Oriented and Entity-Relationship Modelling. OOER '95, London, UK, Springer-Verlag (1995) 270–281
6. Plessers, P., De Troyer, O., Casteleyn, S.: Understanding ontology evolution: A change detection approach. Web Semantics: Science, Services and Agents on the World Wide Web. **5**(1) (2007) 39–49
7. Stojanovic, L.: Methods and tools for ontology evolution. PhD thesis, University of Karlsruhe (2004)
8. Javed, M., Abgaz, Y., Pahl, C.: A pattern-based framework of change operators for ontology evolution. In: On the Move to Meaningful Internet Systems: OTM 2009 Workshops. Volume 5872 of Lecture Notes in Computer Science. (2009) 544–553
9. Ren, X., Shah, F., Tip, F., Ryder, B.G., Chesley, O.: Chianti: a tool for change impact analysis of java programs. SIGPLAN Not. **39** (October 2004) 432–448
10. Abgaz, Y., Javed, M., Pahl, C.: Empirical analysis of impacts of instance-driven changes in ontologies. In: On the Move to Meaningful Internet Systems: OTM 2010 Workshops. Lecture Notes in Computer Science. (2010)
11. Qin, L., Atluri, V.: Evaluating the validity of data instances against ontology evolution over the semantic web. Information and Software Technology. **51**(1) (2009) 83–97
12. Garousi, V., Briand, L., Labiche, Y.: Analysis and visualization of behavioral dependencies among distributed objects based on uml models. In Nierstrasz, O., Whittle, J., Harel, D., Reggio, G., eds.: Model Driven Engineering Languages and Systems. Volume 4199 of Lecture Notes in Computer Science. (2006) 365–379
13. Cox, L., Harry, D., Skipper, D., Delugach, H.S.: Dependency analysis using conceptual graphs. In: In Proceedings of the 9th International Conference on Conceptual Structures, ICCS 2001, Springer (2001)