Composite Ontology Change Operators and their Customizable Evolution Strategies

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

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

Abstract. Change operators are the building blocks of ontology evolution. Elementary, composite and complex change operators have been suggested. While lower-level change operators are useful in terms of fine-granular representation of ontology changes, representing the intent of change requires higher-level change operators. Here, we focus on higher-level composite change operators to perform an aggregated task. We introduce composite-level evolution strategies. The central role of the evolution strategies is to preserve the intent of the composite change with respect to the user's requirements and to reduce the change operational cost. Composite-level evolution strategies assist in avoiding the illegal changes or presence of illegal axioms that may generate inconsistencies during application of a composite change. We discuss few composite changes along with the defined evolution strategies as an example that allow users to control and customize the ontology evolution process.

Keywords: Composite Change, Composite-level Evolution Strategies, Ontology Change Operator Framework, Semantic Ontology Evolution.

1 Introduction

Ontologies can support tasks ranging from capturing the conceptual knowledge, architecture and process models/patterns to the organisation and traceability of digital content and other information artifacts. Ontologies are essential for knowledge sharing. In such domains, ontologies can convey useful semantic information for content managers, ontology engineers, domain experts etc. to understand and process. However, information systems are always subject to change and ontology change management can pose challenges. The reason for such changes can be the changes in the domain, the specification, the conceptualization or any combination of them [1]. A change in an ontology may originate from a domain knowledge expert, a user of the ontology or a change in the application area [2]. Some changes are about the introduction of new classes, removal of outdated classes and changes in the structures and the description of classes. These change may affect the semantic or structural validity of the data [3].

Many ontology evolution tasks cannot be done by a single atomic change operation and require higher-level change operations. Based on different perspectives, different combinations of atomic change operations can be utilized to

perform a composite task. While performing an ontology change using atomic change operations, users can utilize different atomic-level evolution strategies [4] to resolve any inconsistencies. Such evolution strategies are essential at each level of granularity. In this paper, we focus on the composite change operators of our layered change operator framework [5] and present evolution strategies suitable for composite changes. Central features of our presented work are:

- Elementary, composite and complex change operators have been proposed in literature [4, 6, 8]. This indicates that the effectiveness of an ontology change is significantly dependent on the granularity, how the change operators are combined and the extent of their effect on the ontology. Thus, a coherent treatment of the change operators and their effect on the consistency at each level of granularity becomes vital.
- Composite change operators and the composite-level evolution strategies can reduce the overall operational cost and to achieve a specific intent of change. Each evolution strategy is most appropriate in one specific situation.

The paper is structured as follows: A description of our empirical study and the layered change operator framework is given in Section 2. In Section 3, we discuss higher-level composite change operators. In Section 4, we present higher-level evolution strategies suitable for the specific composite change operations. A brief evaluation is given in Section 5. We end with related work and conclusions.

2 Layered Ontology Change Operator Framework

We studied the evolution of domain ontologies empirically in order to investigate the relationships between generic and domain-specific changes and to determine common patterns of change. Based on our observation of common changes, we proposed a layered framework of change operators:

- *level one* captures atomic changes which are elementary tasks,
- *level two* captures aggregated changes to represent composite tasks,
- *level three* captures domain-specific change patterns.

In this paper, we focus on level two *composite change operators* only. As a case study [5], the domains *University Administration* and *Database Systems* were taken into consideration. The former is selected as it represents an organisation involving people, organisational units and processes. The latter is a technical domain that can be looked at from different viewpoints for instance, being covered in a course or a textbook on a subject. We observed that the changes in the database system can be identified by taking different perspectives into account. In teaching, the course content evolves almost every year introducing new concepts, theories and languages. In publishing, new database books in the area appear every couple of years resulting in addition of new chapters, merging or removal of existing chapters and changing of the structure of the topics within and among chapters. In industry, new technologies and languages are emerging. These changes result both in schema and instance-level changes.

3 Composite Ontology Change Operators

Atomic level operationalisation and representation of ontology changes can only describe the addition or deletion of an ontology element. The semantics of applied change(s) are missing from such representation. Level two *composite change operators* fill this gap. They can be utilized to perform aggregated generic tasks and present the intent of the applied changes more explicitly. Composite change operators are identified by grouping atomic change operations of level one.

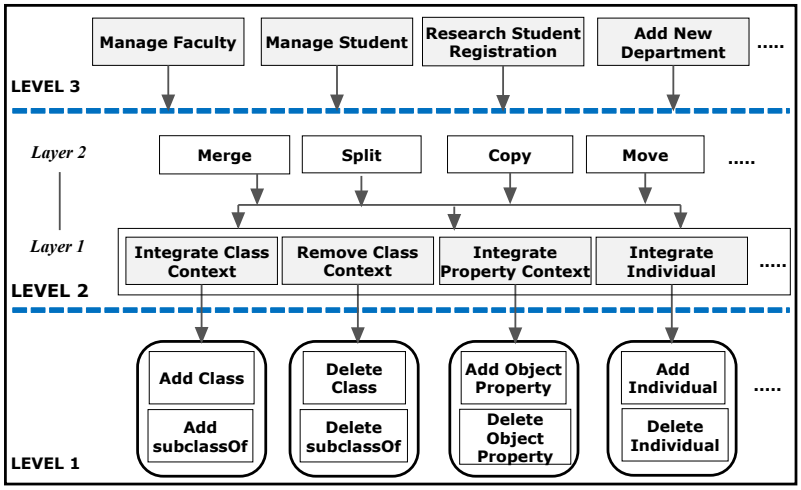

Fig. 1. Layered Architecture of Ontology Change Operations (*University Ontology*)

Composite change operations come in two layers (Figure 1). First layer of composite change operations includes group of those atomic change operations that in general are executed together. For example, *"Remove Class Context"* which not only deletes a class from the hierarchy, but also deletes all its roles[1]. To delete a single class *"faculty"* in a university ontology, deleting the class from the hierarchy is not sufficient. Before we delete the class, we have to delete all axioms that contain class *"faculty"* as a parameter, such as deleting from the domain and the range of properties like *"isSupervisorOf"* or *"hasPublication"* etc. In addition, we need to either delete its subclasses or link them to the parent class in order to keep the ontology consistent. If an ontology engineer wants to merge two or more classes, the operation requires operators higher than the integrate/remove class context. In such a case, composite change operations from layer two can be used. *"merge classes"*, *"move classes"* and *"pull up property"* are the examples of layer two composite change operations. In this paper, we primarily focus on layer two composite change operators. For simplicity, we refer them as "composite change operators" here onwards.

[1] a role is an ontology axiom by which an entity (`Class`, `Property`, `Individual`) is attached to another entity of the domain ontology.

While atomic change operators are useful for fine-granular representation of ontology changes, representing the *intent* of the changes at the atomic level is not feasible. If a class x is removed from a parent y and is attached (as a subclass) to another class s, the semantics behind such change (at atomic level) only refers to a change of the class hierarchy for class x, i.e. `Move class` (`x`, `y`, `s`). However, if we represent that class s is actually a superclass of y, the semantics behind such a change will refer to a `Pull up class` (`x`, `y`, `s`) change.

There is no agreed standard set of composite change operations that one could based on. It is obvious and also mentioned in research [4, 8] that one can combine different atomic level change operations in order to construct new composite changes. Thus, providing an exhaustive list of composite change operations is not feasible. In our current work, we adopted the composite change operations and their definitions from [4] and are given in Table 1.

Table 1. List of composite change operations and their definitions

Composite Change	**Description**
Split class (x, (x_1, x_2))	Split a class x into two newly created sibling classes x_1 and x_2.
Merge class ((x_1, x_2), x)	Merge two existing classes x_1 and x_2 into one newly created class x.
Pull up class (x, x_1)	Pull class x up in its class hierarchy and attach it to all parents of its previous parent x_1.
Pull up class (x)	Pull class x up in its class hierarchy and attach it to all parents of all its previous parents.
Pull down class (x, x_1)	Pull class x down in its class hierarchy and attach it as a child to its previous sibling class x_1.
Pull down class (x)	Pull class x down in its class hierarchy and attach it as a child to all its previous sibling classes.
Move class (x, x_1)	Detach class x from its previous superclass and attach it as a subclass to a class x_1 (which previously was not a direct/indirect super/sub-class of class x).
Group classes (x, (x_1, x_2))	Create a common parent class x for sibling classes x_1 and x_2 and transfer the common properties to it.
Add generalisation class (x, x_1)	Add a new class x between x_1 and all its super classes.
Add specialization class (x, x_1)	Add a new class x between x_1 and all its subclasses.
Pull up property (p, x_1, x_2)	Pull a property p up in the class hierarchy and attach it to the superclass x_2 of its previous domain/range class x_1.
Pull down property (p, x_1, x_2)	Pull a property p down in the class hierarchy and attach it to subclass x_2 of its previous domain/range class x_1.

4 Evolution Strategies for Composite Change Operators

For higher-level ontology change representation, we introduce composite-level evolution strategies that support ontology engineers by suggesting the evolution strategies that may be most appropriate in a specific situation.

As, a composite change is a combination of different atomic level change operations, inconsistent states may emerge during the application of a composite

change operation. In such cases, different sets of atomic change operations can be additionally utilized, along with the change request, to achieve the consistent state of the domain ontology. However, each solution may lead to a distinct consistent ontology version. Here, the term *consistent state* not only refers to a structural consistency but also a semantic consistency [3]. For example, if a class `ResearchStudent` is split into two sibling classes `PhDStudent` and `MSByReseachStudent`, the change will have a direct impact on associated individuals, subclasses and properties (Figure 2). As soon as `ResearchStudent` is

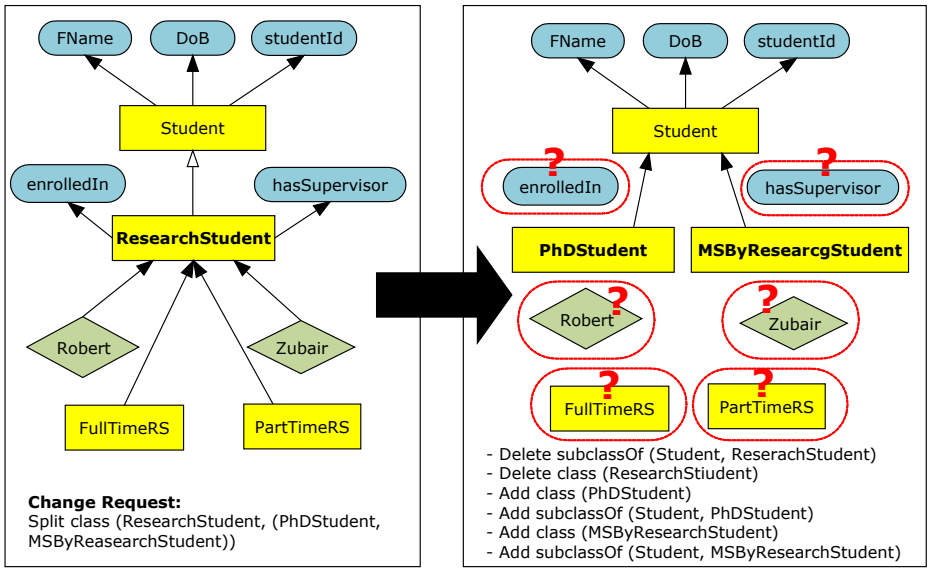

Fig. 2. Split class - Resolution Point

deleted from the class hierarchy,

- the individuals of class `ResearchStudent` will lose their type (rdf:type),
- the subclasses of `ResearchStudent` will become direct subclasses of owl:Thing
- properties will lose their domain/range.

Leaving the individuals without a type and properties without a domain/range has a semantic impact [3] and the purpose of the split change is not achieved. These issues must be resolved in such a way that the intent of the composite change is acquired. One possibility here is to utilize the (atomic level) evolution strategies proposed in [4]. However, the proposed evolution strategies deal with the structural inconsistencies only and do not consider semantic inconsistencies. Based on the given evolution strategies, the orphaned individuals will either be i) deleted, ii) unique individuals will be deleted or iii) reconnected to the parent of the deleted class (i.e., `Student`). The subclasses will either be i) deleted, ii) attached to the parent class `Student` or iii) reconnected to the owl:Thing and the properties will either be i) deleted or ii) left without any domain/range [4]. According to the semantics of the *split* change request, the individuals, subclasses

and properties of the deleted class `ResearchStudent` must be preserved and attached to the replacement classes (i.e., `PhDStudent` and `MSByReseachStudent`). There are many ways to resolve this resolution point[2].

Different sets of change operations can be utilized to resolve the composite change resolution point. Each set of change operations will lead to a distinct ontology version. It would be too restrictive if we enforce one single solution for a resolution point. Such solution may fulfill requirements of one ontology engineer but not of others. A flexible mechanism is required here that allows ontology engineers to resolve the resolution points based on their requirements and needs. This can be achieved by using *composite-level evolution strategies*. The evolution strategies are to preserve the semantics of a composite change in such a way that domain ontology stays consistent.

To identify different sets of *evolution strategies*, we started by exploring the set of composite changes that can be applied to a domain ontology. Later, we inspected what structural and semantic impact each composite change may have on ontology entities and in which circumstances the domain ontology may become inconsistent. Here, we examined the involved ontology entities of the composite change and their inter- and intra-dependencies to other neighborhood ontology entities. We scrutinized the changes that may lead to the subsequent changes. Next step was to identify the resolution points. For each resolution point, we defined evolution strategies and each evolution strategy represents one possible way to resolve the (structural/semantic) consistency issue.

Split class: In the example for composite change `Split class`, the newly added sibling classes `PhDStudent` and `MSByResearchStudent` adopt roles from class `ResearchStudent`. Thus, deleted relationships (axioms) of `ResearchStudent` must be preserved and re-attached to the newly added classes. The question arise here *how to re-attach the deleted roles?* Different users may adopt different approaches to resolve this resolution point. A user can either

- distribute the deleted roles of class `ResearchStudent` among the replacement classes, OR
- re-attach the roles to one of the replacement class, OR
- re-attach roles to both the replacement classes, OR
- do nothing.

Here, a role can be re-attached to two or more classes using the "or" or "and" property. For example, we can re-attach the property `hasPublication` as `domainOf(hasPublication) = PhDStudent or MSByReseachStudent`, i.e. if an individual instantiate the property `hasPublication`, the individual is either a PhD student or a MS by research student.

Now, we present five different defined composite changes, their structural and semantic impacts, resolution points and proposed evolution strategies to resolve such resolution points. The list below is not exhaustive.

[2] Resolution points are the branch points where application of different sets of change operations may lead to different sets of consistent ontology versions [12].

Pull up class (X, C):

- *Structural impact:* The class `X` is pulled up in the hierarchy and became a sibling of its previous parent class `C`.
- *Semantic impact:* Individuals of `X` are not individual of `C` anymore (inference).
- *Resolution point:* Given, a class `X` is disjoint to class `C`,
  $type(\texttt{I}) = \texttt{X} \Rightarrow \neg\ type(\texttt{I}) = \texttt{C}$ (and vice versa)
  Composite change may make ontology inconsistent if a predefined disjointness exists among the siblings of class `C` (i.e., `X` and `C` become disjoint classes in version 2). In such case, any instantiation of property `P` whose domain/range consists class `C`, by any individual of class `X`, is not valid anymore. If such instantiations of `P` exist, ontology will become inconsistent.
- `Evolution Strategies:`
  – If the property `P` does not fit for the class `C` anymore, user can delete the instantiation of the property `P` for the individuals of `C`, OR
  – If the individuals of class `X` can still be individuals of `C`, user can delete the disjointness between the classes `C` and `X`, OR
  – If individuals of class `X` cannot be considered as individuals of class `C` anymore however the property `P` is still valid for the individuals of class `C`, in such case user can explicitly add class `X` as domain/range of the property `P` i.e., `domainOf(P) = C or X`.

Pull down class (A, B):

- *Structural impact:* The class `A` is pulled down in the class hierarchy and became a child of its previous sibling class `B`.
- *Semantic impact:* Individuals of class `A` are individuals of class `B` as well.
- *Resolution point:* Given two classes `A` and `B`,
  $subclassOf(A, B) \Rightarrow \neg\ disjointClasses(A, B)$
  Domain ontology will become inconsistent if the classes `A` and `B` were disjoint to each other before the execution of the composite change. In such case, individuals of class `A` cannot be referred as individuals of class `B`.
- `Evolution Strategies:` In order to resolve the resolution point, user may
  – delete the disjointness between class `A` and `B`.

Merge classes ((C1, C2), X):

- *Structural impact:* classes `C1` and `C2` are replaced by one single class `X`.
- *Semantic impact:* classes `C1` and `C2` are merged into one single class `X` and the relationships (axioms) of classes `C1` and `C2` (that had to be adopted by the class `X`) become un-attached.
- *Resolution point:* The newly added class `X` adopts relationships from the classes `C1` and `C2`. Thus, the deleted relationships (axioms) of classes `C1` and `C2` must be preserved and re-attached to the newly added class `X`.
- `Evolution Strategies:` To resolve the resolution point, user can either
  – aggregate all the deleted roles of classes `C1` and `C2` to the replacement class `X`, OR
  – aggregate selected roles of classes `C1` and `C2` to the replacement class `X`.

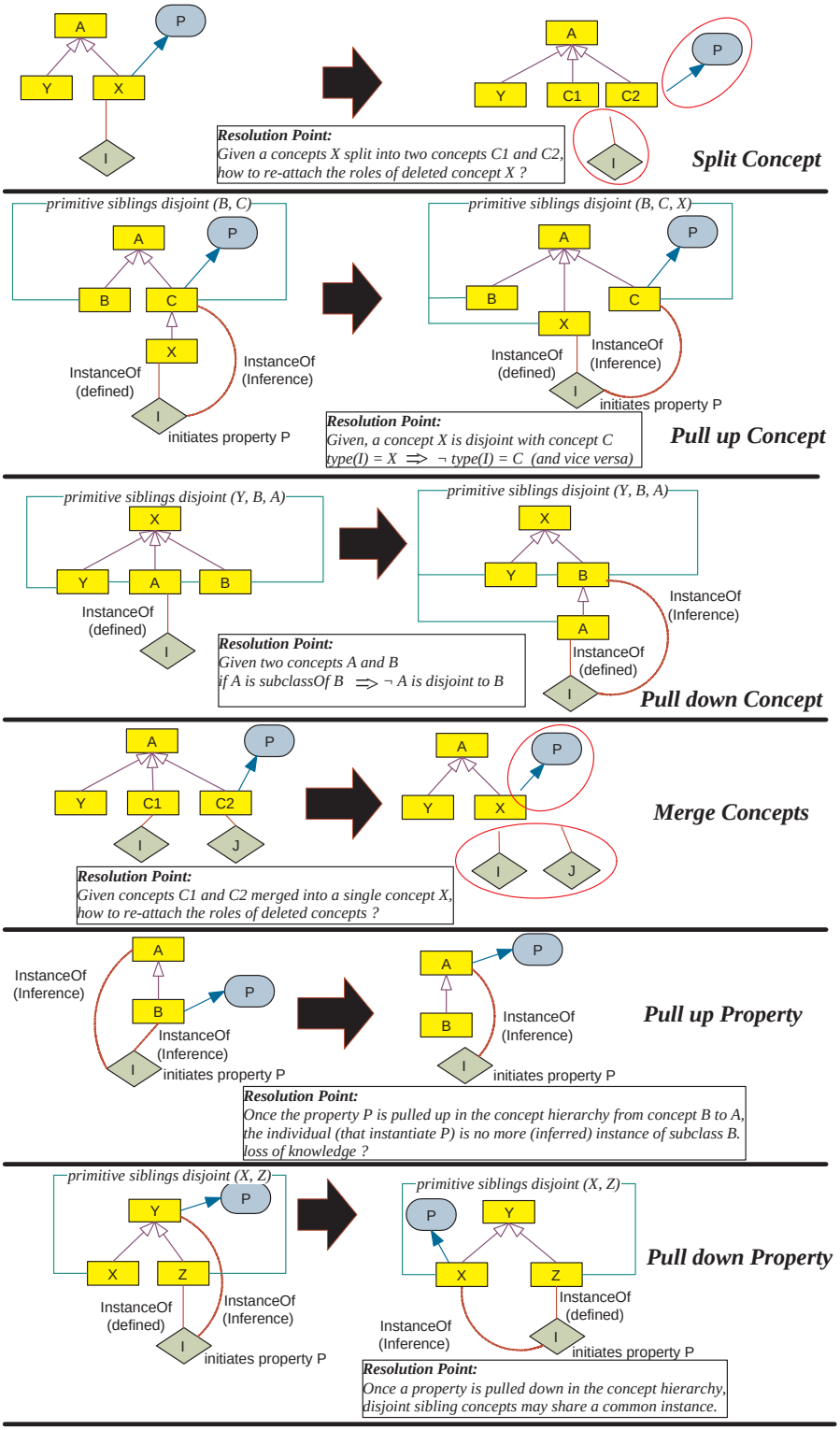

Fig. 3. Composite Changes and their Resolution Points

Pull up property (P, A, B):

- *Structural impact:* The property `P` is attached to the parent class `A` of its earlier domain/range class `B`.
- *Semantic impact:* Earlier, the individuals that instantiate property `P`, were inferred as individuals of class `B` as well as individuals of class `A` (due to subclass hierarchy). After replacing the domain/range of the property `P` (i.e., class `B`) by the parent class `A`, the individuals will be inferred as individuals of class `A` but not of class `B`.
- *Resolution point:* Earlier (inferred) individuals of class `B` (through instantiation of property `P`) are not valid anymore.
- `Evolution Strategies:` In cases, where a user want to make sure that there is no loss of knowledge, i.e., all earlier inferred individuals of child class `B` may still be recognized, user can
  - assert the individuals explicitly as defined individuals of class `B`.

Pull down property (P, X, Y):

- *Structural impact:* The property `P` is attached to the child class `X` of its earlier domain/range class `Y`.
- *Semantic impact:* Earlier, individuals that instantiate property `P` could be inferred as individuals of class `Y` only. After replacing the domain/range of the `P` (i.e., class `Y`) by the child class `X`, the individuals will be inferred as individuals of class `X` as well as of class `Y` (due to subclass hierarchy).
- *Resolution point:* Given,
  $type(\texttt{I}) = \texttt{Z} \wedge siblingClasses(\texttt{X}, \texttt{Z}) \wedge disjointClasses\ (\texttt{X}, \texttt{Z})$
  if, `I` $instantiates$ `P` $\Rightarrow type(\texttt{I}) = \texttt{X}$.
  This unsatisfied disjointness rule (i.e., two disjoint classes cannot share a common individual).
- `Evolution Strategies:` In order to resolve resolution point, a user can
  - remove the disjointness between class `X` and its sibling classes. In such case, an (inferred/defined) individual of `X`'s sibling class, that instantiate property `P`, will also be inferred as individual of class `X`, OR
  - where disjointness between class `X` and its sibling classes is desired, a user can delete the instantiation of the property `P` by the individuals of disjoint sibling classes of class `X`. In such case, the (inferred/defined) individuals of `X`'s sibling classes will no longer be inferred as individuals of class `X`.

5 Evaluation

We evaluated the composite change operators along with the proposed evolution strategies based on their change operational cost. The change operational cost has been evaluated in terms of no. of step to be performed and the time required performing the specified steps. To do so, we selected `split` and `merge` composite change operations along with evolution strategies (Table 2).

Table 2. List of change operations

No.	Change
1	Split class (ResearchStudent, (MSByResearchStudent, PhDResearchStudent)), *Strategy: Split the Roles*
2	Split class (ResearchStudent, (MSByResearchStudent, PhDResearchStudent)), *Strategy: Attach to both classes*
3	Split class (ResearchStudent, (MSByResearchStudent, PhDResearchStudent)), *Strategy: Attach to one classes*
4	Merge classes ((MSByResearchStudent, PhDResearchStudent), ResearchStudent), *Strategy: Aggregate all roles*
5	Merge classes ((MSByResearchStudent, PhDResearchStudent), ResearchStudent), *Strategy: Aggregate selected roles*

Comparison between two levels (i.e., atomic and composite) of change operations, in terms of number of required steps and the average operational time (in seconds), is given in Figure 4. The learning affects (of different users) on the performance of the operational time has been considered and omitted in our controlled experiment. Usage of evolution strategies and pattern-driven data entry

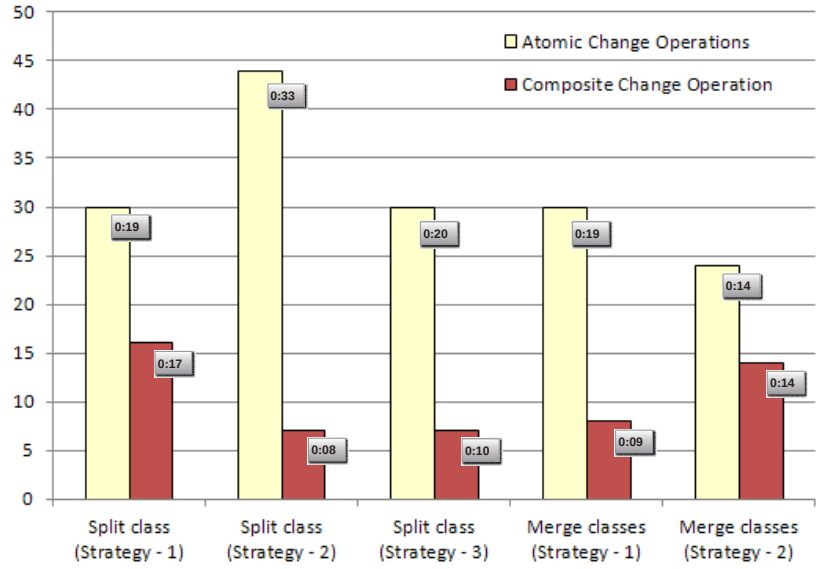

Fig. 4. Atomic/Composite change operations - steps and avg. operational time (in sec.)

forms (for performing composite change operations) reduce the required evolution effort. For example, in case of `Merge classes - Strategy 1` change operation, user need to take eight (8) steps in comparison to thirty (30) atomic change operations. The biggest difference was seen in case of `Split class` change operation where the selected strategy was to join the roles to both the newly added classes (change 2). The result is fairly understandable as in case of atomic change

operators, user need to attach each role one after the other and hence increases the number of required steps. More the roles (to be attached), more the steps it requires. On the other hand, in case of composite change operator, user only need to select the appropriate evolution strategy and all the roles will automatically be attached to the newly added split classes. Hence, increase or decrease of roles did not have any effect on the number of required steps.

6 Related Work

Representation of ontology changes using higher-level change operations was first proposed by Stojanovic [4] and Klein [8]. The change operators are classified into *elementary*, *composite* and *complex*. The identified change operators focus on generic and structural changes only lacking domain-specificity and abstraction. Taking into account the different perspectives of the users and domain experts, our proposed framework defines an additional layer of domain-specific change patterns - which are neglected by the lower-level compositional change operators addressed in the literature. Recently, some researchers have also focused on detection/discovery of higher-level ontology changes [13–15].

Evolution strategies were first proposed by Stojanovic [12] where she considered them as solutions for keeping the ontology consistent at each resolution point. In order to automate the process, Stojanovic also introduced *advanced evolution strategies* that automatically combine available elementary evolution strategies. Such evolution strategies includes *structure-driven*, *process-driven*, *instance-driven* and *frequency-driven* evolution strategies. The proposed evolution strategies resolve structural consistency issues only and thus, are useful and applicable to atomic level change operations only. In order to evolve the domain ontology according to the needs of the user using composite change operators, composite-level evolution strategies are necessary.

7 Conclusion

Ontologies are dynamic entities and will continue to evolve. Scalability beyond manual evolution and change support is required for both stand-alone ontologies that live over very long periods of time and also ontologies used in conjunction with other information systems that themselves can trigger change. Supporting this process by automated management of change and its analysis is a solution.

Most of the ontology evolution tasks cannot be done using a single atomic change operation and thus requires combination of them. We focused on the operationalisation of ontology changes using higher-level change operators and proposed composite-level evolution strategies. These higher-level evolution strategies can deal with the semantic inconsistent states that may emerge during or after the applied composite change. As different users may construct their own composite changes, evolution strategies can be customized to resolve any resolution point. Such higher-level evolution strategies were missing in the past and need to be incorporated in any ontology editing framework.

Acknowledgement

This research is supported by the Science Foundation Ireland (Grant 07/CE/I1142) as part of the Centre for Next Generation Localisation (`www.cngl.ie`) at Dublin City University.

References

1. Noy, N.F., Klein, M.: Ontology evolution: Not the same as schema evolution. In: Journal of Knowledge and Information Systems. Volume 6(4). (2004) 328–440
2. Liang, Y., Alani, H., Shadbolt, N.: Ontology change management in protégé. In: Proceedings of AKT DTA Colloquium, Milton Keynes, UK. (2005)
3. 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
4. Stojanovic, L.: Methods and tools for ontology evolution. PhD thesis, University of Karlsruhe (2004)
5. Javed, M., Abgaz, Y.M., Pahl, C.: A pattern-based framework of change operators for ontology evolution. In: On the Move to Meaningful Internet Systems: OTM Workshops. Volume 5872 of LNCS., Springer (2009) 544–553.
6. Palma, R., Haase, P., Corcho, O., Gomez-Perez, A.: Change representation for owl 2 ontologies. In: Proceedings of the sixth international workshop on OWL: Experiences and Directions (OWLED). (2009)
7. Gruhn, V., Pahl, C., Wever, M.: Data Model Evolution as Basis of Business Process Management. In: 14th International Conference on Object-Oriented and Entity Relationship Modelling O-O ER95. Springer-Verlag (LNCS Series). (1995)
8. Klein, M.: Change Management for Distributed Ontologies. PhD thesis, Vrije University Amsterdam (2004)
9. Boyce, S., Pahl, C.: The development of subject domain ontologies for educational technology systems. Educational Technology and Society **10**(3) (2007) 275–288
10. Holohan, E., Melia, M., McMullen, D., Pahl, C.: The generation of e-learning exercise problems from subject ontologies. Sixth International Conference on Advanced Learning Technologies. (2006) 967–969.
11. Pahl, C., Giesecke, S., Hasselbring, W.: An ontology-based approach for modelling architectural styles In: European Conference on Software Architecture ECSA'2007. Springer-Verlag (LNCS Series) (2007) 60–75
12. Stojanovic, L., Maedche, A., Motik, B., Stojanovic, N.: User-driven ontology evolution management. Volume 6 of LNCS., springer (2002) 285–300
13. Papavassiliou, V., Flouris, G., Fundulaki, I., Kotzinos, D., Christophides, V.: On detecting high-level changes in RDF/S KBs. In: 8th International Semantic Web Conference. Volume 5823 of LNCS., Springer (2009) 473–488
14. Groner, G., Staab, S.: Categorization and recognition of ontology refactoring pattern. Technical Report 09/2010, Institut WeST, Univ. Koblenz-Landau (2010)
15. Javed, M., Abgaz, Y.M., Pahl, C.: Graph-based discovery of ontology change patterns. In: Joint Workshop on Knowledge Evolution and Ontology Dynamics (EvoDyn): ISWC Workshops. (2011)