NATIONAL TECHNICAL UNIVERSITY OF ATHENS

SCHOOL OF ELECTRICAL AND COMPUTER ENGINEERING

DIVISION OF COMPUTER SCIENCE

Managing Evolution in Web Data through Complex Changes

PhD Thesis

of

Theodora Galani
Electrical and Computer Engineer
National Technical University of Athens (2010)

Athens, November 2021

ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ
ΣΧΟΛΗ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ
ΤΟΜΕΑΣ ΤΕΧΝΟΛΟΓΙΑΣ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΥΠΟΛΟΓΙΣΤΩΝ

Διαχείριση Εξέλιξης σε Δεδομένα Ιστού με τη Χρήση Σύνθετων Αλλαγών

Διδακτορική Διατριβή

της

Θεοδώρας Γαλάνη
Διπλωματούχου Ηλεκτρολόγου Μηχανικού & Μηχανικού Υπολογιστών
Εθνικού Μετσόβιου Πολυτεχνείου (2010)

Συμβουλευτική Επιτροπή : Ι. Βασιλείου
Β. Καντερέ
Ι. Σταύρακας

Εγκρίθηκε από την επταμελή εξεταστική επιτροπή την 2^{α} Νοεμβρίου 2021.

Ιωάννης Βασιλείου Ομοτ. Καθηγητής Ε.Μ.Π.	Βασιλική Καντερέ Επικ. Καθηγήτρια Ε.Μ.Π.	Ιωάννης Σταύρακας Ερευνητής Α Ε.Κ. Αθηνά
Δημήτριος Τσουμάκος Αναπλ. Καθηγητής Ε.Μ.Π.	Παναγιώτης Βασιλειάδης Καθηγητής Παν. Ιωαννίνων	Γεώργιος Παπαστεφανάτος Ερευνητής Β Ε.Κ. Αθηνά
	Γεώργιος Στάμου Αναπλ. Καθηγητής Ε.Μ.Π.	

Αθήνα, Νοέμβριος 2021

. . .

Theodora Galani

Electrical and Computer Engineer, PhD, N.T.U.A.

. . .

Θεοδώρα Γαλάνη

Διδάκτωρ Ηλεκτρολόγος Μηχανικός και Μηχανικός Υπολογιστών Ε.Μ.Π.

Abstract

The increasing amount of information published on the web poses new challenges for data management. A central issue concerns evolution management. Data published on the web frequently change, as errors may need to be fixed or new knowledge has to be incorporated. Data consumers need to know *what* changed among versions, as well as *how* and *why*. Revisiting past data snapshots and versions may not be enough for tracking and understanding the semantics of data evolution. Such an activity may require a search that moves backwards and forwards in time, spread across disparate parts of a database, and perform complex queries on the semantics of the changes that modified the data, a task which may be even more intensive for large datasets. In our view, for understanding data evolution changes should be treated as *first-class-citizens*. This means that *human-readable*, *semantically rich* changes are supported, along with any *relations* between them. Treating changes as first-class-citizens poses several challenges regarding *modeling*, *defining*, *detecting* and *querying* changes. In this thesis, we study these directions and work upon two basic standards for web data: RDF and XML.

First, we propose our approach on modeling, defining and detecting changes in the context of RDF(S) knowledge bases. Overall, the proposed approach offers expressiveness and flexibility in terms of evolution interpretation. The proposed *complex changes* provide additional information for interpreting past data, via capturing relations between changes and allowing interpreting evolution in multiple ways.

Specifically, we proposed modeling and supporting simple and complex changes, as well as any relations among them, for interpreting evolution on RDF(S) knowledge-bases. Simple changes are fine-grained and application/data-agnostic changes, while complex changes are coarse-grained and application/data-specific changes. Furthermore, we formally defined an intuitive, user-friendly language, based on change semantics for defining complex changes. We formally defined the language syntax, via EBNF specification, as well as the language semantics. Moreover, we presented a detection algorithm for the proposed complex change definition language. The dynamics model followed is to detect changes between dataset versions. Therefore, the ultimate goal of defining complex changes is identifying complex change instances between dataset versions, via the complex change detection process. Also, the correctness of the proposed implementation with

respect to the language semantics is presented. Finally, we extensively evaluated the proposed approach both qualitatively and experimentally. The qualitative evaluation showed the added value of our approach compared to related works. The experimental evaluation showed the complex change language expressiveness and the detection performance. The proposed language is proven to be adequate in expressing useful changes and facilitating user in analyzing evolution. The response time of the detection process is examined in terms of increasing dataset size. The experimental evaluation is performed over both artificial and real data, proving the effectiveness of our approach.

Second, we propose a query language for querying both data versions and change structures in the context of semistructured XML data. This work builds upon *evo-graph*, a model that captures evolving data along with changes, and *evoXML*, an XML representation of evo-graph.

Specifically, we formally defined *evo-path*, an XPath extension for performing time-aware and change-aware queries on evo-graph. Evo-path allows querying both data history and change structure in a uniform way, supporting *temporal*, *evolution* and *causality* queries. We presented the evo-path syntax, we defined evo-path formal semantics and we presented an implementation based on a formal translation of evo-path into equivalent XPath expressions over evoXML. Also, we implemented and experimentaly evaluated the basic concepts of evo-graph in the *C2D framework*, using XML technologies. The space efficiency of evoXML is examined for various configurations, as well as the performance of the *reduction* process, the process for generating a snapshot holding under a specific time instance from evo-graph.

Keywords: change modeling, change definition language, change detection, RDF(S), querying data evolution, XML, XPath

Περίληψη

Ο αυξανόμενος όγκος πληροφοριών που δημοσιεύονται στο διαδίκτυο δημιουργεί νέες προκλήσεις για τη διαχείριση δεδομένων. Ένα κεντρικό ζήτημα αφορά τη διαχείριση της εξέλιξης. Τα δεδομένα που δημοσιεύονται στον ιστό συχνά αλλάζουν, καθώς πιθανά σφάλματα ενδέχεται να πρέπει να διορθωθούν ή να ενσωματωθεί νέα γνώση. Οι χρήστες των δεδομένων πρέπει να γνωρίζουν *τί* άλλαξε μεταξύ των εκδόσεων, καθώς και *πώς* και *γιατί*. Συνεπώς, η ανάγκη για τη διατήρηση των εκδόσεων δεδομένων και τον προσδιορισμό των αλλαγών γίνεται εμφανής.

Συγκεκριμένα, η επανεξέταση προηγούμενων στιγμιότυπων και εκδόσεων δεδομένων μπορεί να μην είναι αρκετή για την παρακολούθηση και κατανόηση της σημασιολογίας της εξέλιξης των δεδομένων. Μια τέτοια δραστηριότητα μπορεί να απαιτεί μια αναζήτηση που κινείται προς τα πίσω και προς τα εμπρός στο χρόνο, εξαπλώνεται σε διαφορετικά μέρη μιας βάσης δεδομένων και εκτελεί σύνθετα ερωτήματα σχετικά με τη σημασιολογία των αλλαγών που τροποποίησαν τα δεδομένα, μια εργασία που μπορεί να είναι ακόμη πιο απαιτητική για μεγάλα σύνολα δεδομένων. Μια τυπική προσέγγιση για τη διαχείριση των αλλαγών είναι ο υπολογισμός των διαφορών μεταξύ των εκδόσεων δεδομένων (Berners-Lee and Connolly (2004) [4]; Volkel et al. (2005) [59]; Franconi et al. (2010) [21]; Zeginis et al. (2011) [65]; Noy and Musen (2002) [43]; Klein (2004) [33]; Marian et al. (2001) [38]; Cobena et al. (2002) [17]; Wang et al. (2003) [61]). Ωστόσο, αυτή η προσέγγιση οδηγεί σε μια μηχανιστική αναπαράσταση των αλλαγών που δεν παρέχει καμία διαισθητική ερμηνεία σχετικά με τη σημασιολογία των αλλαγών ή πιθανών σχέσεων μεταξύ τους. Επομένως, η πρόθεση ή η αιτία μιας αλλαγής δεν μπορεί να αποτυπωθεί, και πιο σημαντικά το γεγονός ότι μια αλλαγή μπορεί να είναι μέρος μιας μεγαλύτερης αλλαγής σε ένα σύνολο δεδομένων.

Κατά την άποψή μας, για την κατανόηση της εξέλιξης των δεδομένων οι αλλαγές θα πρέπει να αντιμετωπίζονται ως *πρώτης τάξης πολίτες*. Αυτό σημαίνει ότι *κατανοητές από τον άνθρωπο*, *σημασιολογικά πλούσιες* αλλαγές υποστηρίζονται, μαζί με τυχόν *σχέσεις* μεταξύ τους. Η αντιμετώπιση των αλλαγών ως πρώτης τάξης πολίτες θέτει αρκετές προκλήσεις σχετικά με τη *μοντελοποίηση*, τον *ορισμό*, τον *εντοπισμό* και την *επερώτηση* αλλαγών.

Όσον αφορά στη *μοντελοποίηση αλλαγών,* οι αλλαγές θα πρέπει να μοντελοποιηθούν ως οντότητες που διατηρούν *σημασιολογικά* και *δομικά* χαρακτηριστικά. Προς αυτήν την κατεύθυνση, δύο βασικά

ζητήματα πρέπει να ληφθούν υπόψη: ο *βαθμός λεπτομέρειας* των αλλαγών και η *σημασιολογία* των αλλαγών. Όσον αφορά το βαθμό λεπτομέρειας, οι *λεπτομερείς* αλλαγές έχουν το πλεονέκτημα να περιγράφουν θεμελιακές αλλαγές, ενώ οι *συνοπτικές* αλλαγές παρέχουν περισσότερη σημασιολογία και περιεκτικότητα, ομαδοποιώντας τις θεμελιακές αλλαγές σε λογικές μονάδες. Όσον αφορά τη σημασιολογία, οι αλλαγές που *αγνοούν την εκάστοτε εφαρμογή και δεδομένα* περιγράφουν μεταβολές που ενδέχεται να εμφανιστούν σε ένα συγκεκριμένο μοντέλο αναπαράστασης, ενώ οι αλλαγές που *είναι συγκεκριμένες για την εκάστοτε εφαρμογή και δεδομένα* αναπαριστούν μεταβολές που καθορίζονται από τον χρήστη και ταιριάζουν σε συγκεκριμένα σενάρια χρήσης. Το μοντέλο αλλαγών που θα πρέπει να ακολουθείται πρέπει να είναι όσο το δυνατόν πιο ευέλικτο και εκφραστικό.

Όσον αφορά στον *ορισμό αλλαγών*, η υποστήριξη αλλαγών που καθορίζονται από τον χρήστη είναι απαραίτητη προϋπόθεση για αλλαγές που αφορούν συγκεκριμένες εφαρμογές/δεδομένα, οι οποίες εμπλουτίζουν σημαντικά την ερμηνεία της εξέλιξης. Επιπλέον, επιτρέπονται πολλαπλές ερμηνείες της εξέλιξης σε μια συγκεκριμένη εφαρμογή ή σύνολο δεδομένων, δεδομένου ότι οι επιμελητές ή οι χρήστες των δεδομένων μπορεί να ενδιαφέρονται για διαφορετικά τμήματα της εξέλιξης ή να έχουν διαφορετική κατανόηση για μεταβολές που έχουν εφαρμοστεί. Επίσης, η υποστήριξη αλλαγών που καθορίζονται από τον χρήστη καθιστά τους ορισμούς των αλλαγών επαναχρησιμοποιήσιμους, διευκολύνοντας περαιτέρω τη διαδικασία ορισμού νέων αλλαγών. Σε αυτή την περίπτωση, η ιεραρχική δομή που δημιουργείται καθώς μια αλλαγή χτίζεται πάνω σε άλλες καταδεικνύει σχέσεις και εξαρτήσεις μεταξύ τους. Προς αυτήν την κατεύθυνση, απαιτείται μια ειδική γλώσσα για τον ορισμό των αλλαγών.

Όσον αφορά στον *εντοπισμό αλλαγών*, όπως ήδη συζητήθηκε, μια τυπική προσέγγιση για το χειρισμό των αλλαγών μεταξύ των εκδόσεων συνόλου δεδομένων είναι ο υπολογισμός των διαφορών μεταξύ τους. Ομοίως, καθώς νέες εκδόσεις δεδομένων κυκλοφορούν περιοδικά, στιγμιότυπα κατανοητών από τον άνθρωπο, σημασιολογικά πλούσιων αλλαγών μπορούν να εντοπιστούν μεταξύ τους. Ειδικά, για την περίπτωση μιας ειδικής γλώσσας για τον ορισμό των αλλαγών, θα πρέπει να διερευνηθεί πώς αυτοί οι ορισμοί των αλλαγών μπορούν να χρησιμοποιηθούν για την παρακολούθηση σχετικών μεταβολών μεταξύ εκδόσεων δεδομένων. Επομένως, οι κατάλληλοι αλγόριθμοι για τον εντοπισμό στιγμιότυπων αλλαγών ως πρώτης τάξης πολίτες μεταξύ εκδόσεων δεδομένων θα πρέπει να διερευνηθούν.

Όσον αφορά στην *επερώτηση αλλαγών*, επερωτήσεις που αφορούν την εξέλιξη δεδομένων μπορούν επίσης να παρέχουν πληροφορία σχετικά με τον τρόπο που άλλαξαν τα δεδομένα. Εκτός από τα χρονικά ερωτήματα [26][49] που επιστρέφουν εκδόσεις ιστορικών δεδομένων, δεδομένου ότι οι αλλαγές μοντελοποιούνται ως πρώτης τάξης πολίτες, μπορούν επίσης να αξιοποιηθούν στα πλαίσια

επερωτήσεων. Ιδανικά, επερωτήσεις που αφορούν την εξέλιξη δεδομένων θα πρέπει να βασίζονται τόσο σε δεδομένα όσο και σε αλλαγές. Οι αλλαγές, όπως και τα δεδομένα, μπορούν να εμφανιστούν στο σώμα μιας επερώτησης για να εκφράσουν σύνθετες συνθήκες, όπως το γεγονός ότι μια οντότητα έχει τροποποιηθεί με συγκεκριμένο τρόπο, ή μπορούν να επιστραφούν από την επερώτηση προκειμένου να ανακτηθούν επακριβώς τα στιγμιότυπα των αλλαγών που έχουν επηρεάσει συγκεκριμένα δεδομένα. Προς αυτήν την κατεύθυνση, ένα μοντέλο που καταγράφει τόσο τις εκδόσεις δεδομένων όσο και τις αλλαγές είναι απαραίτητη προϋπόθεση για την έκφραση τέτοιων ερωτημάτων. Επίσης, θα πρέπει να διερευνηθεί μια γλώσσα επερωτήσεων με συγκεκριμένα δομικά στοιχεία για να υποστηριχθούν τόσο χρονικές συνθήκες όσο και συνθήκες που αφορούν τις αλλαγές.

Στην παρούσα διατριβή, μελετάμε τις παραπάνω κατευθύνσεις και εργαζόμαστε πάνω σε δύο βασικά πρότυπα για δεδομένα στον ιστό: το RDF [34] και το XML [7].

Οι κατευθύνσεις της μοντελοποίησης, ορισμού και εντοπισμού αλλαγών έχουν μελετηθεί στο πλαίσιο των βάσεων γνώσεων RDF(S). Οι μέθοδοι που προτείναμε και τα αποτελέσματα που παράχθηκαν δημοσιεύθηκαν στο [23], ενώ στο [22] και [24] δημοσιεύτηκαν πρωταρχικές εργασίες.

Συγκεκριμένα, προτείναμε τη μοντελοποίηση και την υποστήριξη απλών και σύνθετων αλλαγών, καθώς και τυχόν σχέσεων μεταξύ τους, για την ερμηνεία της εξέλιξης σε βάσεις γνώσεων RDF(S). Οι απλές αλλαγές είναι λεπτομερείς αλλαγές και αγνοούν την εκάστοτε εφαρμογή και δεδομένα, πράγμα που σημαίνει ότι δεν περιλαμβάνουν άλλες αλλαγές και η σημασιολογία τους ταιριάζει με το μοντέλο δεδομένων RDF. Οι σύνθετες αλλαγές είναι συνοπτικές αλλαγές που αφορούν την εκάστοτε εφαρμογή και δεδομένα, πράγμα που σημαίνει ότι επιδεικνύουν δομή και σημασιολογία κατάλληλη για κάθε συγκεκριμένη εφαρμογή ή σύνολο δεδομένων. Αν και η μοντελοποίηση αλλαγών κατανοητών από τον άνθρωπο, μέσω θεμελιωδών αλλαγών και ομαδοποιήσεων αυτών εξετάζεται στη βιβλιογραφία (Klein (2004) [33]; Stojanovic (2004) [57]; Papavasileiou et al. (2013) [45]; Roussakis et al. (2015) [53]), ενώ άλλοι (Plessers, De Troyer and Casteleyn (2007) [47]; Roussakis et al. (2015) [53]) θεωρούν επίσης αλλαγές οριζόμενες από τον χρήστη, οι σχέσεις και οι εξαρτήσεις μεταξύ σύνθετων αλλαγών δεν υποστηρίζονται στις υπάρχουσες προσεγγίσεις.

Επιπλέον, ορίσαμε τυπικά μια διαισθητική, φιλική προς το χρήστη γλώσσα, βασισμένη στη σημασιολογία των αλλαγών για τον ορισμό σύνθετων αλλαγών. Συγκεκριμένα, ορίσαμε τυπικά το συντακτικό της γλώσσας, μέσω της προδιαγραφής EBNF, καθώς και τη σημασιολογία της γλώσσας. Όλες οι έννοιες της γλώσσας παρουσιάζονται λεπτομερώς και αρκετά παραδείγματα επεξηγούν αυτές τις έννοιες. Εν γένει, οι σύνθετες αλλαγές ορίζονται μέσω μοτίβων πάνω από απλές αλλαγές και ήδη ορισμένες σύνθετες αλλαγές.

Επιπρόσθετα, παρουσιάσαμε έναν αλγόριθμο εντοπισμού στιγμιοτύπων αλλαγών για την προτεινόμενη γλώσσα ορισμού σύνθετων αλλαγών. Το δυναμικό μοντέλο που ακολουθείται αφορά στον εντοπισμό αλλαγών μεταξύ των εκδόσεων ενός συνόλου δεδομένων. Επομένως, ο απώτερος στόχος του ορισμού σύνθετων αλλαγών είναι ο εντοπισμός στιγμιοτύπων σύνθετων αλλαγών μεταξύ των εκδόσεων δεδομένων, μέσω της διαδικασίας εντοπισμού σύνθετων αλλαγών. Παρουσιάζουμε λεπτομερώς τον αλγόριθμο εντοπισμού, καθώς και την ορθότητα του προτεινόμενου αλγορίθμου σε σχέση με τη σημασιολογία της γλώσσας.

Η προτεινόμενη προσέγγιση αξιολογήθηκε εκτενώς τόσο ποιοτικά όσο και πειραματικά. Στην ποιοτική αξιολόγηση, η προσέγγισή μας συγκρίνεται με συναφείς εργασίες ως προς βασικά χαρακτηριστικά και κατά πόσο αυτά υποστηρίζονται, δείχνοντας την προστιθέμενη αξία της προσέγγισής μας. Στην πειραματική αξιολόγηση, εξετάζεται η εκφραστικότητα της γλώσσας ορισμού σύνθετων αλλαγών και η επίδοση του αλγορίθμου εντοπισμού. Αξιολογείται εάν τα προτεινόμενα χαρακτηριστικά της γλώσσας είναι επαρκή για την έκφραση χρήσιμων αλλαγών και πώς οι σύνθετες αλλαγές διευκολύνουν τον χρήστη στην ανάλυση της εξέλιξης. Επίσης, ο χρόνος απόκρισης της διαδικασίας εντοπισμού εξετάζεται σε σχέση με την αύξηση του μεγέθους του συνόλου δεδομένων. Η αξιολόγηση πραγματοποιείται τόσο σε τεχνητά όσο και σε πραγματικά δεδομένα, αποδεικνύοντας την αποτελεσματικότητα της προσέγγισής μας.

Συνολικά, η προτεινόμενη προσέγγιση προσφέρει εκφραστικότητα και ευελιξία ως προς την ερμηνεία της εξέλιξης. Η προτεινόμενη μοντελοποίηση σύνθετων αλλαγών παρέχει πρόσθετες πληροφορίες για την ερμηνεία παρελθοντικών δεδομένων, επιτρέπει την ερμηνεία της εξέλιξης με πολλαπλούς τρόπους, ενώ η αποτύπωση σχέσεων μεταξύ σύνθετων αλλαγών είναι ένα επιπλέον χαρακτηριστικό που εμπλουτίζει την εκφραστικότητα των σύνθετων αλλαγών.

Η κατεύθυνση της επερώτησης αλλαγών έχει μελετηθεί στο πλαίσιο των ημιδομημένων δεδομένων XML. Η προσέγγιση που προτείναμε βασίζεται στο *evo-graph* [55], ένα μοντέλο που καταγράφει εξελισσόμενα δεδομένα μαζί με τις αλλαγές, και το *evoXML* [56], μια XML αναπαράσταση του evo-graph. Αξίζει να σημειωθεί ότι στο evo-graph οι αλλαγές μοντελοποιούνται χρησιμοποιώντας απλές (εκεί ονομάζονται βασικές) και σύνθετες αλλαγές. Οι μέθοδοι που προτείναμε δημοσιεύθηκαν στο [25], ενώ μια πρώτη αξιολόγηση σχετικά με το evo-graph δημοσιεύθηκε στο [44].

Συγκεκριμένα, ορίσαμε τυπικά την *evo-path*, μια επέκταση της XPath [51] για την εκτέλεση επερωτήσεων βάσει του χρόνου και των αλλαγών στο evo-graph. Η Evo-path επιτρέπει την επερώτηση τόσο του ιστορικού των δεδομένων όσο και της δομής των αλλαγών με ενιαίο τρόπο, εκμεταλλευόμενη τις αλλαγές, ανακτώντας και να συσχετίζοντας δεδομένα που διαφορετικά είναι άσχετα μεταξύ τους. Υποστηρίζονται *ερωτήματα χρονικά*, *ερωτήματα εξέλιξης* και *ερωτήματα*

αιτιότητας. Παρουσιάσαμε το συντακτικό της evo-path, ορίσαμε τυπικά τη σημασιολογία της evo-path και παρουσιάσαμε μια υλοποίηση που βασίζεται σε μια τυπική μετάφραση της evo-path σε ισοδύναμες εκφράσεις XPath πάνω στο evoXML.

Τέλος, υλοποιήσαμε και αξιολογήσαμε πειραματικά τις βασικές έννοιες του evo-graph στο *πλαίσιο «C2D»*, χρησιμοποιώντας τεχνολογίες XML. Συγκεκριμένα, αξιολογήσαμε την αποδοτικότητα σε χώρο του evoXML για διάφορες περιπτώσεις. Αξιολογήσαμε επίσης την επίδοση της *διαδικασίας παραγωγής ενός στιγμιότυπου* που αντιστοιχεί σε μια συγκεκριμένη χρονική στιγμή από το evo-graph, σε σχέση με το μέγεθος του αρχείου evoXML. Η αξιολόγηση που πραγματοποιήθηκε έδειξε ποιοι παράγοντες που χαρακτηρίζουν τα δεδομένα επηρεάζουν το μέγεθος του evoXML και τη διαδικασία παραγωγής ενός στιγμιότυπου.

Λέξεις Κλειδιά: μοντελοποίηση αλλαγών, γλώσσα ορισμού αλλαγών, εντοπισμός αλλαγών, RDF(S), επερώτηση εξελισσόμενων δεδομένων, XML, XPath

Acknowledgements

I would like to wholeheartedly thank my supervisor Prof. Yannis Vassiliou for his continuous support and encouragement. I also appreciate the contributions of Dr. Yannis Stavrakas and Dr. George Papastefanatos throughout our collaboration. I would like also to thank Assistant Prof. Verena Kantere for serving as a member of my advisory committee.

I would like to thank Prof. Timos Sellis for being my supervisor at the early years of my PhD studies and later for being a member of my advisory committee.

I would like to thank Associate Prof. Dimitrios Tsoumakos, Prof. Panos Vassiliadis and Associate Prof. Georgios Stamou for serving as members of my examination committee.

I would like to thank the anonymous reviewers of the publications stemming from this thesis, who helped me shape this work with their valuable comments.

I also thank the Special Account for Research Funding (E.L.K.E.) of N.T.U.A for supporting me with a scholarship for my PhD studies.

Lastly, I would like to thank my family, and especially my sister Mary, for their love, support and encouragement during all the years of my studies.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1. Motivation

The increasing amount of information published on the web poses new challenges for data management. A central issue concerns evolution management. Data published on the web frequently change, as errors may need to be fixed or new knowledge has to be incorporated. Data consumers need to know *what* changed among versions, as well as *how* and *why*. Thus, the need for maintaining data versions and identifying changes becomes evident.

In particular, revisiting past data snapshots and versions may not be enough for tracking and understanding the semantics of data evolution. Such an activity may require a search that moves backwards and forwards in time, spread across disparate parts of a database, and perform complex queries on the semantics of the changes that modified the data, a task which may be even more intensive for large datasets. A typical approach for handling changes is computing diffs between dataset versions (Berners-Lee and Connolly (2004) [4]; Volkel et al. (2005) [59]; Franconi et al. (2010) [21]; Zeginis et al. (2011) [65]; Noy and Musen (2002) [43]; Klein (2004) [33]; Marian et al. (2001) [38]; Cobena et al. (2002) [17]; Wang et al. (2003) [61]). However, this approach leads to a *machine-readable* representation of changes that does not provide any intuition about change semantics or possible relations between them. Therefore, the intention or the cause of a change cannot be captured, and more importantly the fact that a change may be part of a larger change in a dataset.

In our view, for understanding data evolution changes should be treated as *first-class-citizens*. This means that *human-readable*, *semantically rich* changes are supported, along with any *relations* between them. Treating changes as first-class-citizens poses several challenges regarding *modeling*, *defining*, *detecting* and *querying* changes.

Modeling changes. Changes should be modeled as entities that retain *semantic* and *structural* characteristics. Towards this direction, two basic issues must be taken into consideration: *granularity* of changes and *semantics* of changes. As for granularity, *fine-grained* changes have the advantage of describing primitive changes, while *coarse-grained* changes provide more semantics and conciseness by grouping primitive changes into logical units. As for semantics, *application/data-agnostic* changes describe modifications that may appear in a specific representation model, while *application/data-specific* changes represent user-defined changes that suit on specific use-case scenarios. The change model to be followed should be as much flexible and expressive as possible.

Defining changes. Supporting user-defined changes is a prerequisite for application/data-specific changes, which significantly enrich evolution interpretation. Even more, multiple interpretations of evolution on a specific application or dataset are allowed, since data curators or consumers may be interested in different parts of evolution or have different understanding on applied modifications. Also, supporting user-defined changes makes their definitions reusable, further facilitating the process of defining new changes. In this case, the hierarchical structure created while a change is built on top of others demonstrates relations and dependencies among them. Towards this direction, a dedicated language for defining changes is needed.

Detecting changes. As already discussed a typical approach for handling changes among versions is computing diffs between them. Similarly, as new dataset versions are periodically released, instances of human-readable and semantic rich changes may be detected between them. Especially, in case of a dedicated language for defining changes, it should be investigated how these change definitions may be used for tracking relevant modifications between versions. Therefore, appropriate algorithms for detecting change instances as first-class-citizens among dataset versions should be investigated.

Querying changes. Querying data evolution may also provide insights on how data changed. Apart from temporal queries [26][49] that return historical data versions, since changes are modeled as first-class-citizens, they can be also exploited in terms of querying. Ideally, querying data evolution should be based on data as much as on changes. Changes, like data, can appear in the query body to express complex conditions, like the fact that an entity has been modified in a specific manner, or can be returned by the query in order to retrieve explicit change instances that may have affected specific data. Towards this direction, a model that captures both data versions and changes is a prerequisite in order to express such

queries. Also, a query language with specific constructs to support both temporal and change based conditions should be investigated.

1.2. Contributions

In this thesis, we study the above directions and work upon two basic standards for web data: RDF [34] and XML [7]. The contributions of this thesis are summarized below.

The directions of modeling, defining and detecting changes have been studied in the context of RDF(S) knowledge bases. The methods that we proposed and the results obtained were published in [23], while in [22] and [24] a preliminary and a visionary work were published respectively.

1. We proposed modeling and supporting simple and complex changes, as well as any relations among them, for interpreting evolution on RDF(S) knowledge-bases. Simple changes are fine-grained and application/data-agnostic changes, meaning that they do not comprise of other changes and their semantics suit to the RDF data model. Complex changes are coarse-grained and application/data-specific changes, meaning that they demonstrate structure and semantics suitable to each specific application or dataset. Although modeling human-readable changes via primitive changes and groupings of them is considered in literature (Klein (2004) [33]; Stojanovic (2004) [57]; Papavasileiou et al. (2013) [45]; Roussakis et al. (2015) [53]), while others (Plessers, De Troyer and Casteleyn (2007) [47]; Roussakis et al. (2015) [53]) consider user-defined changes as well, relations and dependencies among complex changes are not supported in any of the already existing approaches.

2. We formally defined an intuitive, user-friendly language, based on change semantics for defining complex changes. Specifically, we formally defined the language syntax, via EBNF specification, as well as the language semantics. All language concepts are presented in detail and several examples illustrate these concepts. Overall, complex changes are defined via patterns over simple changes and already defined complex changes.

3. We presented a detection algorithm for the proposed complex change definition language. The dynamics model followed is to detect changes between dataset versions. Therefore, the ultimate goal of defining complex changes is identifying

complex change instances between dataset versions, via the complex change detection process. We present in detail the detection algorithm, as well as the correctness of the proposed implementation with respect to the language semantics.

4. We extensively evaluated the proposed approach both qualitatively and experimentally. In qualitative evaluation, our approach is compared to related works regarding basic features and characteristics they support, showing the added value of our approach. In experimental evaluation, complex change language expressiveness and detection performance are examined. It is evaluated whether the proposed structures are adequate in expressing useful changes and how complex changes facilitate user in analyzing evolution. Also, the response time of the detection process is examined in terms of increasing dataset size. The evaluation is performed over both artificial and real data, proving the effectiveness of our approach.

Overall, the proposed approach offers expressiveness and flexibility in terms of evolution interpretation. The proposed modeling of complex changes provides additional information for interpreting past data, allows interpreting evolution in multiple ways, while capturing relations among complex changes is an additional feature that enriches the complex changes' expressivity.

The direction of querying changes has been studied in the context of semistructured XML data. The approach that we proposed builds upon previous work done in [55][56], regarding *evo-graph*, a model that captures evolving data along with changes, and *evoXML*, an XML representation of evo-graph. It is worth noting that in evo-graph changes are modeled using simple (there named as *basic*) and complex changes. The methods that we proposed were published in [25], while some first evaluations regarding evo-graph were published in [44].

5. We formally defined *evo-path*, an XPath [51] extension for performing time-aware and change-aware queries on evo-graph. Evo-path allows querying both data history and change structure in a uniform way, taking advantage of changes in order to retrieve and relate data that are otherwise irrelevant to each other. *Temporal*, *evolution* and *causality* queries are supported. We presented the evo-path syntax, we defined evo-path formal semantics and we presented an implementation based on a formal translation of evo-path into equivalent XPath expressions over evoXML.

6. We implemented and experimentally evaluated the basic concepts of evo-graph in the *C2D framework*, using XML technologies. Specifically, we evaluated the space

efficiency of evoXML for various configurations. We also evaluated the performance of the *reduction* process, the process for generating a snapshot holding under a specific time instance from evo-graph, with respect to the size of the evoXML file. The evaluation performed indicated which factors that characterize the data affect the evoXML size and the reduction process.

1.3. Thesis Outline

The remainder of this thesis is structured as follows: Chapter 2 presents the related work which is categorized in two main pillars: modeling and detecting changes in knowledge bases, and modeling and querying data evolution in semistructured data. Chapter 3 presents our work in defining and detecting complex changes in RDF(S) knowledge bases. Specifically, we present: the proposed simple and complex changes concepts, the formal specification of our complex change definition language, the relevant detection algorithm and the details of the evaluation performed. Chapter 4 presents our work in querying evolving data and changes in XML. Specifically, evo-path syntax, semantics and implementation are presented, as well as our first experiments on evo-graph. Chapter 5 concludes this thesis and presents future research steps.

Chapter 2

Related Work

2.1. Modeling and Detecting Changes in Knowledge Bases

2.1.1. Machine-readable changes

A number of works focus on computing the differences between knowledge bases in terms of insertions and deletions, which are not concise neither intuitive. They focus on machine-readable changes and some of them introduce useful properties for the proposed deltas.

In Berners-Lee and Connolly (2004) [4] the problem of comparing two RDF graphs, generating a set of differences, and updating a graph from a set of differences is discussed. Generating differences between RDF graphs is straightforward when all nodes are named: the delta between the RDF graphs is a pair of insertions and deletions. When not all nodes are named, finding the largest common sub-graphs becomes a case of the graph isomorphism problem. However, in a wide set of practical cases, one can efficiently generate a delta. When named and unnamed nodes are mixed, but none of the unnamed nodes is very far from a named node, the unnamed nodes can be identified by being in context with a named node, via a path, so that differences are expressed by giving this local context and the related changes. Furthermore, the authors propose an update ontology for representing differences between RDF graphs, in terms of insertions and deletions. A patch file format provides a way to uniquely identify what changed, as well as whether it was added or subtracted. Also, two forms of difference information are discussed: the context-sensitive “weak” deltas and the context-free “strong” deltas. A weak delta gives enough information to apply it to exactly the graph it was computed from, but a strong delta specifies the changes in a context independent manner. One advantage of a strong delta is that one can take a delta from any true knowledge base change and apply it to a subset knowledge base, and the result will be true. Strong deltas eliminate the possible failure of a patch to find the appropriate points in the RDF graph at which to apply the changes. The proposed methodology for generating strong deltas applies

only on graphs which are well-labeled directly with URIs or indirectly with functional properties or inverse functional properties.

In Volkel et al. (2005) [59] an RDF-centric versioning approach and a relevant implementation called SemVersion are presented. SemVersion provides structural and semantic versioning for RDF models and RDF based ontology languages, like RDFS. Two algorithms for generating diffs are proposed, together with an RDF representation for the diffs, while the implemented system supports several operations (like commit, branch, merge, etc.) inspired by the well-known versioning system in the developer community CVS. Regarding the diff algorithms, the first one is for computing a structural diff as a triple-set-based difference between two models. Two triple sets, of added and removed statements, are computed. A speciality of RDF is the usage of blank nodes, adding complexity to the diff computation. If a user commits a new model and later requests a diff, the system cannot tell whether two blank nodes are equal or different. They have by definition no globally unique identifier. Blank node enrichment is proposed to overcome this problem by uniquely identifying blank nodes. It creates an "enriched model" from a normal model by introducing a new property, whose value plays the role of an inverse functional property like in OWL. Blank nodes should only have one such property value assigned. This unique URI makes them globally addressable, while they remain formally blank nodes in the RDF model. All existing RDF semantics are still valid. The second diff algorithm is for computing a semantic diff, given an RDF based ontology language. In this case, the semantically inferred triples are also taken into account while computing the diff. Thus, a language specific reasoner or rules should be available for the calculation. Regarding the representation of RDF diffs, the following approach is proposed: a triple is made addressable by reification, sets of triples are represented as rdfs:Bags, leading to a trivial triple set ontology. A full RDF diff contains a triple set of added and a triple set of removed statements, and additionally the blank node enrichment statements have to be added.

In Franconi et al. (2010) [21], the scenario considered is where a knowledge base (expressed in some logical formalism) might evolve over time and thus different versions have to be maintained, while users of the knowledge base should be able to access, not only any specific version, but also the differences between two given versions of the knowledge base. To address this problem a general semantic framework is proposed. The notion of semantic difference between knowledge bases plays a central role. The proposed approach, is applicable to a large class of logic-based knowledge representation languages. While restricting to finitely generated propositional logics, it is shown that the semantic framework can be represented syntactically in a particular kind of normal form (ordered complete

conjunctive normal form). This is followed by a generalization, where similar results can be obtained for any syntactic representation (in a finitely generated propositional logic) of the semantic framework. Although the methodology focuses on propositional logic knowledge bases, it can be extended to more expressive languages, such as description logics. Regarding the proposed semantic diff, a number of desired properties are examined. First, the semantic diff highlights the differences in terms of the logical meaning between two knowledge bases. Therefore, although two (propositional) knowledge bases may be syntactically different, if they convey exactly the same meaning (they are logically equivalent), there should be no semantic difference between them. Second, in order to avoid redundancy and to comply with the principle of minimal change, the sentences to be added should be contained in the new version and similarly sentences to be removed should be contained in the previous version. Third, the semantic diff should provide an 'undo' operation when moving from one version of a knowledge base to another, so that one is able to roll back any modification performed. Finally, a unique semantic diff is associated with any two knowledge bases. Regarding the overall framework, the scenario examined is when there are n versions of a knowledge base that need to be stored and a core knowledge base. In order to be able to access any version of the knowledge base, it is sufficient to store the core knowledge base and the semantic diff among the core and each version. The core knowledge base may be selected not to be one of the versions, it can be the 'average' of the versions, i.e. a representation minimizing the overall semantic diff of the core to each of the knowledge bases. Alternatively, several reasons are discussed to consider one of the versions as the core knowledge base.

In Zeginis et al. (2011) [65], several issues on computing deltas over RDF(S) knowledge bases are discussed. Five RDF(S) differential functions are presented, which take into account inferred knowledge and return sets of change operations (add / delete). Namely, explicit (Δe), closure (Δc), dense (Δd), dense & closure (Δdc), and explicit & dense (Δed) differential function are presented. Δe returns the set difference over the explicitly asserted triples, while Δc returns the set difference by also taking into account the inferred triples. In order to focus on small sized deltas, Δd, Δdc and Δed are introduced. Assuming knowledge bases K and K', Δd contains add operations for those triples which are explicit in K′ and do not belong to the closure of K, and delete operations for those triples which are explicit in K and do not belong to the closure of K′. Δd produces the smallest in size set of change operations, but can be applied to transform K to K′ only under certain conditions. For this reason, Δdc and Δed are considered. Δdc resembles to Δd regarding additions and to Δc regarding deletions, while Δed resembles to Δe regarding additions and to Δd regarding deletions. For the proposed deltas containment, size and computational complexity are examined. Regarding change operations, triple addition and deletion are considered in order to transform one knowledge base to

another, while two approaches are proposed regarding their semantics: one plain set-theoretic (Up) and another that involves inference and redundancy elimination (Uir). Under Up semantics, only the explicitly defined triples are taken into account while inferred ones are ignored. Under Uir semantics, change operations incur also side-effects such as redundancy elimination and knowledge preservation: the updated knowledge base will not contain any explicit triple which can be inferred, while preserves as much of the knowledge expressed in the former base as possible. Also, several useful properties of RDF(S) deltas are discussed: semantic identity, non redundancy, reversibility and composition. Semantic identity defines that a delta reports an empty result if its operands are semantically equivalent. Non-redundancy defines that the execution of a delta results in a knowledge base that is always redundancy-free. Reversibility of a delta is a useful property for moving forward and backward across versions. Composition allows composing a number of consecutive deltas and then executing the operations of the composed delta, instead of having to execute each particular delta. Another introduced notion is the correctness of a differential function - change operation semantics pair. It ensures that for any two knowledge bases K and K', starting from K and applying the computed delta via the change operation semantics, the result knowledge base is equivalent to K'. A study on which combinations of differential functions and change operation semantics can be employed to correctly transform a source to a target RDF(S) knowledge base is presented. Finally, the computing time and size of the produced deltas over real and synthetic RDF(S) knowledge bases are experimentally investigated, as well as the impact of the inferential potential of the knowledge base. In this work blank nodes and relevant issues are not examined.

In Noy and Musen (2002) [43] a fixed point algorithm named PROMPTDIFF for detecting ontology change is proposed. It consists of two parts: (1) an extensible set of heuristic matchers and (2) a fixed-point algorithm to combine the results of the matchers to produce a structural diff between two versions. The output of the PROMPTDIFF algorithm is a table which bases on a structural diff, which describes the components of the ontology that have changed from one version to another, and also provides more detailed information on how the components have changed. It is stated whether each component was added, deleted, split, merged, or none of the above, and which is the mapping level of each mapped components, defining whether they are different enough from each other to warrant the user's attention. A mapping level may be unchanged, isomorphic, or changed. Each matcher employs a small number of structural properties of the ontologies to produce matches. The fixed-point step invokes the matchers repeatedly, feeding the results of one matcher into the others, until they produce no more changes in the diff. PROMPTDIFF uses a dependency table to determine the order in which it executes the matchers. It keeps a stack of matchers it still needs to run. It

starts by putting the matchers that do not affect any other matchers at the bottom of the stack and matchers that are not affected by other matchers at the top. Then it executes matcher M at the top of the stack. If M produced changes in the PROMPTDIFF table, the algorithm adds to the stack all the matchers that depend on M, removing duplicates. It runs until the stack is empty. The performance of the algorithm has been evaluated and it correctly identifies 96% of the matches in ontology versions from large projects. Notice that the use of heuristics introduces uncertainty to results. Finally, the knowledge model that is used is compatible with the Open Knowledge Base Connectivity protocol, but the methodology can be applied on other representation formalisms such as RDFS and DAML+OIL.

2.1.2. Human-readable changes

Other works focus on supporting human-readable changes, which are usually distinguished between simple and composite/complex. Some of them propose predefined lists of changes, while others user defined changes.

In Klein (2004) [33], an extension of Noy and Musen (2002) [43] is presented for detecting some of the proposed basic and composite changes. First, the four elements of the proposed framework are presented, as well as how they can interact to solve particular problems: (1) An ontology of basic changes is presented, constituted by a set of operations that can be used to build an ontology in a specific language. The proposed basic changes are generated taking into account the meta model of an ontology language. Namely OWL and OKBC are considered. In this way the generated set of changes is complete with respect to the possible ontology modifications. Every possible change is specified by add and delete operations for each element of the knowledge model, while modify operation is also considered. (2) The notion of complex changes is proposed, where a complex change is composed of multiple basic operations, incorporating some additional knowledge about the change. Two dimensions are used to distinguish between different types of complex operations. On one hand, there is a distinction between atomic and composite operations, on the other hand there is a distinction between simple and rich operations. Composite operations provide a mechanism for grouping a number of basic operations that together constitute a logical entity. Atomic operations are operations that cannot be subdivided into smaller operations. Rich changes are changes that incorporate information about the implication of the operation on the logical model of the ontology. For example, a rich change might specify that the range of a property is enlarged. To identify such changes, the logical theory of the ontology has to be queried. In contrast, simple changes can be detected by analyzing the structure only. (3) The notion of a

transformation set is also presented, which is a set of change operations that specify how a version can be transformed into another. (4) Also, a template for the specification of the relation between different ontology versions is presented. It is worth noting that an RDF-based syntax is discussed. Regarding the extensions to the PROMPTDIFF tool (Noy and Musen (2002) [43]), the first extension uses the mappings produced by PROMPTDIFF as a basis for producing a transformation set, while the second extension is able to detect some composite changes and presents these in a conveniently arranged way to the user. The extended tool uses different visual clues in order to improve the visualization of ontology changes. Finally, another system, OntoView, is also discussed. OntoView implements a comparison mechanism for RDF-based ontologies producing a transformation set. Ontologies are compared at a structural level and additions, deletions and definition changes are distinguished. The algorithm starts with an ontology that is represented in RDF. It first parses a textual representation of the ontology into RDF triples, in order to find the changes in the data model instead of the textual representation, and search for added and deleted statements. Then, it groups the statements into individual class and property definitions of the ontology. The changes in the sets of statements that form these definitions can be analyzed to detect the basic changes from the change ontology, and further aggregated into complex changes.

In Stojanovic (2004) [57] a taxonomy of changes is proposed which comprises of elementary, composite and complex changes, forming a predefined set of changes. Composite changes group elementary changes which appear in the same neighborhood and are generalized by complex changes. Ontology evolution and change semantics have been studied in terms of ontology consistency maintenance. In this work, each change is applied together with a number of generated changes that ensure the ontology consistency. In these terms, the requirements of an ontology evolution management system are outlined, together with an evolution process that satisfies them. Furthermore, the proposed single ontology evolution approach has been extended in order to take into account multiple interdependent ontologies in the context change propagation. Also, a usage-driven approach for change discovery has been presented, where user query and browsing history of an ontology-based application is exploited for the continual adaptation of the ontology to user's needs. The solutions presented have been implemented for KAON ontology. This approach follows an opposite direction on how changes are used, since they are captured as they are applied on the ontology rather than after version generation.

In Plessers, De Troyer and Casteleyn (2007) [47], the Change Definition Language (CDL) is proposed for defining and detecting changes over a version log using temporal queries. The change detection approach presented is in the context of an ontology evolution framework for

OWL DL ontologies. The framework allows ontology engineers to request and apply changes to the ontology they manage, assuring that the ontology and its depending artifacts remain consistent after changes have been applied. The change detection mechanism allows generating a detailed overview of changes that have occurred based on a set of change definitions, while different users may have their own set of change definitions allowing different overviews of the changes and different levels of abstraction. The presented notion of version log keeps track of all the different versions of all concepts ever defined in an ontology, starting from their creation, modifications, until the eventual retirement. Whenever an ontology concept is modified, the version log is updated by creating a new version for the affected concept. CDL allows users to define the meaning of changes in a formal and declarative way. Its syntax is presented in terms of EBNF specification, its semantics are formally defined, and several examples are provided. CDL is based on temporal logic and thus changes are specified in terms of conditions that must hold before and after the appliance of the change (pre-/post-conditions). Together with the version log, the CDL provides the foundation of the change detection approach. The change definitions expressed in the CDL are evaluated as temporal queries on a version log. The outcome is a collection of occurrences of the change definitions. It is worth noting that past tense operators are employed in CDL, expressing cases like 'some time in the past', 'always in the past', 'since', 'previous time', and 'after'. Also, the temporal logic supports two different views on the timeline of a version log. The first view considers the complete timeline as it takes the history of the whole ontology into account, while the second only considers the part of the timeline that belongs to the history of a particular concept. In order to reflect and apply both views, tense operators have been extended by introducing parameterized versions. The approach has been validated by developing two prototype extensions for the Protege ontology editor.

In Auer and Herre (2007) [3] a framework for supporting evolution in RDF knowledge bases is discussed. The approach works on the basis of atomic changes, basically additions or deletions of statements to or from an RDF graph. Such atomic changes are aggregated to compound changes, resulting in a hierarchy of changes, thus facilitating the human reviewing process on various levels of detail. These derived compound changes may be annotated with meta-information, such as the user executing the change or the time when the change occurred. A simple OWL ontology capturing such information is presented. Also, these compound changes can be classified as ontology evolution patterns. Ontology evolution patterns in conjunction with appropriate data migration algorithms enable the automatic migration of instance data in distributed environments. Thus, the evolution of ontologies with regard to higher conceptual levels than the one of statements is allowed. Examples of data

migration algorithms are given. However, neither a detection process, nor a specific language of changes is defined.

In Papavasileiou et al. (2013) [45], a set of predefined high-level changes for RDF(S) knowledge bases (KBs) and an algorithm for their detection are proposed. The presented change language allows the formulation of concise and intuitive deltas. In total 132 changes are defined at the level of RDF(S) constructs, capturing addition, deletion, renaming, move in the hierarchy, change of domain/range etc., that the various constructs (classes, properties etc.) of an RDF(S) KB can undergo. Both basic (i.e., fine-grained changes on individual RDF graph nodes or edges) and composite high-level changes (coarse-grained changes affecting several nodes and/or edges) are considered, while another separate category named heuristic changes is considered too, whose detection conditions require the existence of mappings among data version entities. It is worth noting that operations considered capture changes at both ontology (schema) and instance (data) levels. A set of desired features related to the detection and application semantics of the language of changes is presented. These features are related to both human and machine interpretability. Therefore, the proposed language of changes is guaranteed to (a) be intuitive and capture as accurately as possible the perception and intent of editors regarding the performed changes, (b) be able to handle (describe) any possible change in a unique manner, and, (c) have well-defined formal and consistent detection and application semantics. It is worth noting that the proposed changes verify the properties of completeness and unambiguity, for guaranteeing that every added / deleted triple is consumed by one detected high-level change and that detected high-level changes are not overlapping, respectively. Therefore, any possible change encountered in curated KBs expressed in RDF(S) can be efficiently and deterministically detected in an automated way. Moreover, a change detection algorithm, which is sound and complete with respect to the presented language, is defined. Its correctness and complexity have been studied. Also, the appropriate semantics for executing the deltas expressed in the proposed language of changes are presented, in order to move backwards and forwards in a multi-version repository, using only the corresponding deltas. Finally, the effectiveness and efficiency of the presented algorithms have been experimentally evaluated using real ontologies from the cultural, bioinformatics and entertainment domains.

In Roussakis et al. (2015) [53], an extension of Papavasileiou et al. (2013) [45] is proposed, providing a more generic change definition framework, based on SPARQL [31] queries. The authors acknowledge that different uses (or users) of the data may require a different set of changes being reported, since the importance and frequency of changes vary in different application domains. For this reason, the proposed framework supports both simple and

complex changes. Simple changes are meant to capture fine-grained types of evolution. They are defined at design time and should meet the formal requirements of completeness and unambiguity, which guarantee that the detection process is well-behaved as defined in Papavasileiou et al. (2013) [45]. Complex changes are meant to capture more coarse-grained, changes that are useful for the application at hand. This allows a customized behavior of the change detection process, depending on the actual needs of the application. Complex changes are totally dynamic and defined at run-time. Therefore, it is unrealistic to assume that they will guarantee completeness or unambiguity. As a consequence, in order to avoid any non-determinism in the detection process, complex changes are associated with a priority level. In this way, complex changes may not share common parts or conflict each other. The detection process is based on SPARQL queries (one per defined change) that are provided to the algorithm as configuration parameters. As a result, the core detection algorithm is agnostic to the set of simple or complex changes used, allowing new changes to be defined. Furthermore, to support analysis of the evolution process, an ontology of changes, which allows the persistent representation of the detected changes, is presented. This, in a multi-version repository, allows queries that refer uniformly to both the data and its evolution. The framework has been evaluated experimentally, based on 3 real RDF datasets of different sizes to study the number of simple and complex changes that usually occur in real-world settings, and provide an analysis of their types. Moreover, the evaluation results of the efficiency of the change detection process are presented and the effect of the size of the compared versions and the number of detected changes in the performance of the algorithm are quantified.

In Singh et al. (2018) [54], DELTA-LD approach is presented. Changes are detected and classified between two versions of a linked dataset. The basic contribution is proposing a classification to distinctly identify the resources that have has both their IRI and representation changed and the resources that have had only their IRI changed. The former case is modeled as a renew change, while the latter as a move change, while create, remove, update changes are may also be detected. It is worth noting that an automatic method for selecting resource properties to identify the same resource with different IRIs and similar representation in different versions has been presented. A relevant change model for representing detected changes has also been presented, where a changed resource is accompanied with its added/deleted triples. The accuracy of the proposed approach has been measured and a case study for the automatic repair of broken interlinks using the changes detected by DELTA-LD has been presented.

Finally, Troullinou et al. (2016) [58] focuses on providing metrics for analyzing evolution rather than calculating change entities. This approach aims at giving a high-level overview of

the change process by identifying the most important changes in the ontology. It does not propose specific changes or their detection, but instead considers different metrics of "change intensity". Metrics that take into account the changes that affected each class and its neighborhood are considered, relying on the added and deleted triples among versions. Also, metrics that take into account ontological information related to the importance and connectivity of each class in the different versions are considered. This approach allows understanding the intent (rather than the actions) of the editor and provides a better focusing of the curator analyzing the changes.

2.2. Modeling and Querying Evolution in Semistructured Data

2.2.1. Version-based approaches

Version based approaches mainly focus on aspects regarding managing, storing and querying XML document versions, as well as detecting changes between them.

In Marian et al. (2001) [38], a change-centric method for managing versions in XML data is presented. The authors employ a diff algorithm for detecting changes between two consecutive versions of an XML document. Changes are then represented based on completed deltas and persistent identifiers. Completed deltas are deltas containing additional information and thus can be inverted and composed. Also, the notions of edit scripts and simple deltas are presented, where an edit script is formed as a sequence of specific operations, while a delta is formed as a set of specific operations, avoiding to specify an order of execution as in an edit script. Furthermore, a physical storage policy is proposed, based on storing the current version of the document, a map to handle persistent identifiers and a single XML document containing all forward complete deltas. Based on complete deltas, forward deltas (by pruning of the complete deltas) and backward deltas (by inversion and pruning) can be projected, while backward deltas can be used to reconstruct old versions. Also, since completed deltas are more space consuming than simple deltas, compression methods are examined to reduce redundant storage. Finally, a GUI to display changes to the user is provided.

A similar approach is introduced in Chien, Tsotras and Zaniolo (2001) [13], where instead of edit scripts and deltas calculations, a referenced-based versioning scheme that preserves the logical structure of an evolving document via object references is presented. In this scheme,

new versions hold only the elements that are different from the previous version, whereas a reference is used for pointing to the unchanged elements of past versions. Specifically, each maximum unchanged element (i.e., an element which itself is unchanged but its parent node is changed) is represented by a reference record, referring to the logical location of that unchanged element in the previous version. Additionally, the effectiveness of the proposed approach in supporting queries in the document history and evolution, in addition to the usual content-based queries on version instances, is evaluated. For this, a query taxonomy is presented: (1) temporal selection queries, for retrieving a particular version or successive versions, (2) document evolution and historical queries, focusing on changes between successive versions, (3) structural projection queries, for selecting parts of a document, being a key ingredient for temporal selection or history queries, (4) content-based selection queries, for retrieving all versions that qualify the predicates in the "where" clause. Efficient algorithms for supporting temporal selection (view materialization), structural projection and content-based selection queries are presented, as well as for querying the document evolution history. The proposed representation is shown to be efficient at the transport level, where XML documents are exchanged between remote parties. Finally, the effectiveness of the proposed scheme at the storage level is demonstrated. A usefulness-based page management policy is defined, adapted from transaction-time databases, to ensure efficient temporal clustering between versions.

In Chien et al. (2001) [15], a version management scheme named SPaR is presented, for efficiently storing, retrieving and querying multiversion XML documents. The approach presented is based on durable node numbers and timestamps on the elements of XML documents, to preserve the structure and the history of the document during its evolution. The durable node ids can be used as stable references in indexing the elements and allow the decomposition of the documents in several linked files. The problem of full version reconstruction was studied, while indexing and clustering strategies that assure efficient support for complex queries in the content and document evolution are also examined. A page clustering technique is used to trade off storage efficiency and retrieval efficiency and optimize the overall performance.

In Chien et al. (2002) [16], the authors examine the problem of supporting efficiently complex queries on multiversioned XML documents. For this, they expand SPaR scheme and investigate physical realizations for it. Different storage and indexing strategies are examined so as to optimize SPaR's implementation. The presented methodologies build on the observation that evaluating complex version queries mainly depends on the efficiency of evaluating *partial version retrieval queries*, which retrieve a specific segment of an individual

version instead of the whole version. Specifically, complex path expression queries can be reduced to partial version retrieval queries. Retrieving a segment for a single-versioned XML document is efficient since the target elements are clustered on secondary store by their logical order, but this might not be the case for a multiversion document. For a multiversion document, a segment of a later version may have its elements physically scattered in different pages due to version updates. The authors investigate three indexing schemes to evaluate partial version retrieval in this environment: single multiversion B-Tree, UBCC with a multiversion B-Tree, and UBCC with a multiversion R-Tree, where UBCC is a clustering mechanism standing for Usefulness Based Copy Control.

In Gergatsoulis and Stavrakas (2003) [27], the authors propose Multidimensional XML (MXML), an extension of XML that uses context information to express time and models multifaceted documents. Also, it is demonstrated how MXML can represent a set of basic change operations on XML documents and their corresponding schema.

Several approaches, focus mainly on the detection and less on the representation of the changes between two documents. In Wang et al. (2003) [61], X-Diff algorithm is presented, an effective algorithm on unordered trees that integrates key XML structure characteristics with standard tree-to-tree correction techniques. In Cobena et al. (2002) [17], a linear time diff algorithm for XML data is proposed, which matches unchanged sub-trees between the old and new version. The proposed algorithm was used in the Xyleme project, while it can also be used for HTML documents by XML-izing them. In Leonardi et al. (2005) [36] an XML enabled change detection system, Xandy, is presented. It detects structural and content changes by converting unordered XML documents into relational tuples and using SQL queries. This approach has better scalability compared to X-Diff and comparable quality. In Leonardi and Bhowmick (2005) [35], a relational approach is presented, named Helios, to detect changes in unordered XML documents. The delta quality produced is comparable to Xandy, while for large datasets it is faster than Xandy and X-Diff. Finally, in Chawathe et al. (1996) [12] change detection was studied in the context of hierarchically structured information. The change detection problem was defined as the problem of finding a "minimum-cost edit script" that transforms one data tree to another, while efficient algorithms for computing such an edit script were presented.

2.2.2. Temporal approaches

Typically, temporal approaches enrich data elements with temporal attributes for holding time and extend accordingly query syntax with conditions on the time validity of data.

In Grandi (2004) [29], an annotated bibliography dealing with temporal and evolution aspects in the World Wide Web is presented.

In Amagasa, Yoshikawa and Uemura (2000) [1], a logical data model for representing histories of XML documents is proposed. This model is based on the XPath data model, and extends it in some points: (1) edges have a label that represent their valid time, (2) string-value of text and attribute nodes are modelled as virtual nodes, and (3) text and attribute nodes can contain multiple string-value nodes. Using the proposed data model, it is easy to compute a past state of XML documents, by recursively pruning edges that are not available at specified time and by removing labels from edges. Also, operations based on extending the DOM API are investigated, so that the proposed data model can be implemented. Furthermore, two alternative approaches to the physical implementation of the model are presented, so that data represented in the model are translated to XML documents. The first implementation, named *full*, is for implementing the data model to XML documents as they are, and the second, named *simplified*, is for implementing the data model retaining the original form of XML documents. For both implementations, tags and attributes are used in order to represent temporal information. Finally, temporal queries may be evaluated by taking a snapshot of the XML document and then querying it using a non-temporal query language, while a query language specialized in the proposed temporal XML documents is not provided.

In Rizzolo and Vaisman (2008) [49], the problem of modelling and implementing temporal data in XML is addressed. A data model for tracking historical information in an XML document and for recovering the state of the document as of any given time is proposed. The temporal constraints imposed by the data model are studied, and algorithms for validating a temporal XML document against these constraints are presented, along with methods for fixing inconsistent documents. In the presented model transaction time is used, and containment edges are labelled with temporal intervals. In addition, different ways of mapping the abstract representation into a temporal XML document are discussed. Furthermore, TXPath is introduced, a temporal XML query language that extends XPath 2.0. Both its syntax and semantics are presented. In the second part of the paper, an approach for summarizing and indexing temporal XML documents is presented. In particular it is shown that by indexing *continuous paths*, i.e., paths that are valid continuously during a certain

interval in a temporal XML graph, the query performance is dramatically increased. To achieve this, a new class of summaries is introduced, denoted *TSummary*, that adds the time dimension to the well-known path summarization schemes. Within this framework, two new summaries are presented: *LCP* and *Interval summaries*. The indexing scheme TempIndex integrates these summaries with additional data structures. A query processing strategy based on TempIndex is presented, as well as a type of ancestor-descendant encoding, denoted temporal interval encoding. A persistent implementation of TempIndex is also presented, and a comparison against a system based on a non-temporal path index, and one based on DOM. Finally, a language for updates is sketched, and it is shown that the cost of updating the index is compatible with real-world requirements.

In Gao and Snodgrass (2003) [26], a temporal XML query language, *τ*XQuery, is presented. The authors add valid time support to XQuery by minimally extending the syntax and semantics of XQuery. The goal is to move the complexity of handling time from the user/application code into the *τ*XQuery processor. It is worth noting that the approach may also apply to transaction time queries. *τ*XQuery utilizes the data model of XQuery. The few reserved words added to XQuery indicate three different kinds of valid time queries. *Representational queries* have the same semantics with XQuery, ensuring that *τ*XQuery is upward compatible with XQuery. To write such queries, users have to know the representation of the timestamps and treat the timestamp as a common element or attribute. New syntax for *current* and *sequenced queries* makes these queries easier to write. A current query asks for the information about the current state. Sequenced queries are applied independently at each point in time. To implement *τ*XQuery the stratum approach is adopted, in which a stratum accepts *τ*XQuery expressions and maps each to a semantically equivalent conventional XQuery expression. The XQuery expression is passed to an XQuery engine. Once the XQuery engine obtains the result, the stratum possibly performs some additional processing and returns the result to the user. The advantage of this approach is that it exploits the existing techniques in an XQuery engine, such as the query optimization and query evaluation, while at the same time it does not depend on a particular XQuery engine. The paper focuses on how to perform this mapping, in particular, on mapping *sequenced* queries, which are by far the most challenging. The central issue of supporting sequenced queries (in any query language) is time-slicing the input data while retaining period timestamping. Timestamps are distributed throughout an XML document, complicating the temporal slicing. In those terms, authors propose four optimizations of the initial maximally-fragmented time-slicing approach: selected node slicing, copy-based per-expression slicing, in-place per-expression slicing, and idiomatic slicing, each of which reduces the number of constant periods over which the query is evaluated.

In Wang and Zaniolo (2003) [62], the authors present techniques for managing multiversion documents and supporting temporal queries on such documents. The proposed approach consists of a temporally grouped data model, for representing the successive versions of a document as an XML document, named *V-Document*. Using XML query languages, such as XQuery, complex queries on the content of a particular version can be expressed, as well as on the temporal evolution of the document elements and their contents. Also, the paper discusses the advantages of applying the proposed scheme to XML-published relational data. Finally, efficient implementations of the approach are discussed. In Wang and Zaniolo (2008) [63], the authors further extend and elaborate on the concepts presented in Wang and Zaniolo (2003) [62]. In these terms, a number of case studies are performed, the XChronicler tool is presented, a tool for building V-Documents from the successive versions of arbitrary XML documents, and techniques for the efficient storage and retrieval are discussed.

In Moon et al. (2008) [41], the authors work on the problem of managing the history of database information. Specifically, they propose PRIMA system, which employees two key technologies: The first is a method for publishing the history of a relational database in XML, whereby the evolution of the schema and its underlying database are given a unified representation. This temporally grouped representation makes it easy to formulate sophisticated historical queries on any given schema version using standard XQuery. For this, authors build upon and extend previous work presented in Wang and Zaniolo (2003) [62]. The second key technology is that schema evolution is transparent to the user. A user writes queries against the current schema, while retrieving the data from one or more schema versions. The system then performs the labour-intensive and error-prone task of rewriting such queries into equivalent ones for the appropriate versions of the schema. This feature is particularly important for historical queries spanning over different schema versions. For realizing this feature in PRIMA, Schema Modification Operators (SMOs) are introduced, to represent the mappings between successive schema versions, and an XML integrity constraint language (XIC), to efficiently rewrite the queries using the constraints established by the SMOs. The scalability of the approach has been also tested.

In Dyreson (2001) [19], the *TT*XPath data model and query language are sketched. *TT*XPath extends XPath with support for transaction time. To construct the *TT*XPath data model, snapshots of an XML document are obtained over time. The snapshots are then merged and transaction times are associated with each edge and node. The *TT*XPath query language extends XPath with temporal axes to enable a query to access past or future states, and with

constructs to extract and compare times. *TT*XPath maximally reuses XPath and is fully backwards-compatible with XPath.

2.2.3. Other approaches

An early work is presented in Chawathe, Abiteboul and Widom (1999) [11]. The authors propose a model for representing changes in semistructured data and a language for querying over these changes. The starting point of this work is the Object Exchange Model (OEM), a simple graph-based data model, with objects as nodes and object-subobject relationships represented by labelled arcs. The basic change operations proposed in this graph-based model are node insertion, update of node values, and addition and removal of labelled arcs (node deletion is implicit by unreachability). The proposed change representation model is named DOEM (for Delta-OEM) and uses annotations on the nodes and arcs of an OEM graph to represent changes. Intuitively, the set of annotations on a node or arc represents the history of that node or arc. Representing changes directly as annotations on the affected data, instead of indirectly computing changes as the difference between database states, is an important feature of this approach. For querying over changes, a language based on the Lorel language for querying semistructured data is presented, called Chorel (for Change Lorel). Specifically, the authors extend the concept of Lorel path expressions in order to allow references to the annotations in a DOEM database, resulting in an intuitive and convenient language for expressing change queries in semistructured data. Overall, the user can retrieve information related to the history of nodes and edges, exploiting the change annotations. The implementation of DOEM and Chorel uses a method that encodes DOEM databases as OEM databases and translates Chorel queries into equivalent Lorel queries over the OEM encoding. This scheme has the benefit that there is no need to build from scratch yet another database system. Additionally, the authors present extensions that permit snapshot-based access in the proposed change-based data model. Finally, as an important first application of DOEM and Chorel, a query subscription service that permits users to subscribe to changes in semistructured data was designed and implemented.

In Buneman et al. (2004) [10], the problem of archiving and management of curated databases is addressed in terms of XML and semistructured data. The developed archiving technique is efficient in its use of space and preserves the continuity of elements through versions of the database. The approach uses timestamps, and all versions of the data are merged into one hierarchy where an element appearing in multiple versions is stored only once along with a timestamp. By identifying the semantic continuity of elements and merging them into one

data structure, the proposed technique is capable of providing meaningful change descriptions and allows to easily answer certain temporal queries such as retrieval of any specific version from the archive and finding the history of an element. This is in contrast with approaches that store a sequence of deltas where such operations may require undoing a large number of changes or significant reasoning with the deltas.

In Buneman, Chapman and Cheney (2006) [9], the authors deal with provenance in curated databases. Provenance information concerns the creation, attribution, or version history of data, and in terms of scientific databases it is crucial for assessing data integrity and scientific value. The authors propose and evaluate a practical approach to provenance tracking for data copied manually among databases. It is assumed that all of the user's actions in constructing a target database are captured as a sequence of insert, delete, copy, and paste actions by a provenance-aware application for browsing and editing databases. As the user copies, inserts, or deletes data in her local database *T*, provenance links are stored in an auxiliary provenance database *P*. These links relate data locations in *T* with locations in previous versions of *T* or in external source databases *S*. They can be used to review the process used to construct the data in *T*. In addition, if *T* is also being archived, the provenance links can provide further detail about how each version of *T* relates to the next. An implementation of this technique is presented and the experiments show that although the overhead of a naive approach is fairly high, it can be decreased to an acceptable level using simple optimizations.

Chapter 3

Defining and Detecting Complex Changes on RDF(S) Knowledge Bases

3.1. Introduction

Data published on the web frequently change, as errors may need to be fixed or new knowledge has to be incorporated. As new dataset versions are periodically released, data consumers need to know what changed among versions, as well as how and why.

In this context, we focus on interpreting evolution on RDF(S) knowledge-bases. The Resource Description Framework (RDF) [34] is a recommendation of the World Wide Web Consortium (W3C). In essence, RDF is a graph data model that supports modeling facts about entities in a simple triple format consisting of a *subject*, a *predicate* and an *object*, leading to rich and descriptive directed graphs with semantically labelled edges. Graph nodes represent entities that are identified uniquely by Uniform Resource Identifiers (URIs), this way defining a basis amongst remote agents to publish inherently interlinked datasets (Linked Open Data, LOD). The RDF Schema (RDF(S)) [8] is also a W3C recommendation, that constitutes a simple language that can be used to define a vocabulary (i.e. terms) to be used in an RDF graph. Taxonomies defined in RDF(S) can be used to do some basic inference. The standard recommendation for querying RDF data is SPARQL [31], a graph query language established around the specific features of the RDF model.

In literature, several works (Berners-Lee and Connolly (2004) [4]; Volkel et al. (2005) [59]; Franconi et al. (2010) [21]; Zeginis et al. (2011) [65]; Noy and Musen (2002) [43]; Klein (2004) [33]) have been presented for modeling changes in terms of diffs. These approaches lead to a machine-readable representation of changes based on triple additions and deletions and do not provide any intuition about change semantics or possible relations between them. Therefore, the intention or the cause of a change cannot be captured, and more importantly

the fact that a change may be part of a larger change in a dataset. An ideal approach would compute human-readable, semantically rich changes along with any relations between them.

We argue that for understanding data evolution, changes should be treated as first-class-citizens. Modeling changes should involve fine-grained and application/data-agnostic changes, meaning that they do not comprise of other changes and their semantics suit to the RDF data model, as well as coarse-grained and application/data-specific changes, meaning that they demonstrate structure and semantics suitable to each specific application or dataset. The former changes are named simple changes, while the latter are named complex changes, and are defined on top of simple changes.

Modeling complex changes as user-defined is a prerequisite for being application/data-specific. Even more, this allows multiple interpretations of evolution on a specific application or dataset, since data curators or consumers may be interested in different parts of evolution or have different understanding on the applied modifications. Also, modeling complex changes as user-defined makes their definitions reusable, further facilitating the process of defining new changes. In addition, the hierarchical structure created while a change is built on top of others demonstrates relations and dependencies among them. A complex change may be part of another, may be modeled as a specification or generalization of another, or may cause another. However, complex changes may share common parts without being defined as nested, but having overlaps, providing supplementary interpretation of evolution. Towards this direction, a dedicated language for defining complex changes and a relevant detection algorithm is needed, in order to facilitate the precise modeling and reusability of changes. As a result of the detection process change instances are computed, and then can be queried via standard languages for further analyzing evolution.

In literature, several works (Klein (2004) [33]; Stojanovic (2004) [57]; Plessers, De Troyer and Casteleyn (2007) [47]; Auer and Herre (2007) [3]; Papavasileiou et al. (2013) [45]; Roussakis et al. (2015) [53]; Singh et al. (2018) [54]) focus on human-readable changes. Modeling human-readable changes via primitive changes and groupings of them is also considered by Klein (2004) [33], Stojanovic (2004) [57], Papavasileiou et al. (2013) [45] and Roussakis et al. (2015) [53], while Plessers, De Troyer and Casteleyn (2007) [47] and Roussakis et al. (2015) [53] consider user-defined changes as well. However, relations and dependencies among complex changes are not supported in any of the already existing approaches. Furthermore, we propose a dedicated language based on change semantics for defining complex changes and a relevant detection algorithm.

Overall, the proposed approach offers expressiveness and flexibility in terms of evolution interpretation. The proposed modeling of complex changes provides additional information for interpreting past data, allows interpreting evolution in multiple ways, while capturing relations among complex changes via nesting or overlaps is an additional feature that enriches the complex changes' expressivity. The Chapter main contributions are the following:

- modeling and supporting simple and complex changes, as well as relations among them
- providing an intuitive, user-friendly language based on change semantics for defining complex changes via patterns over simple changes and already defined complex changes,
- providing a detection algorithm for the proposed complex change definition language,
- extensively evaluating the proposed approach both qualitatively and experimentally.

The Chapter outline is as follows: Section 3.2 presents a motivating example of this work. Section 3.3 presents the basic concepts and definitions on simple and complex changes. Section 3.4 presents the syntax, semantics and several examples of the proposed language for defining complex changes. Section 3.5 presents in detail the detection algorithm, the process for identifying complex change instances among dataset versions. Section 3.6 presents the qualitative and experimental evaluation performed.

3.2. Motivating Example

Consider a sample part of DBpedia[1] ontology with information about persons and universities, as in Figure 1. Figure 1 left depicts the initial version (V_{bef}) and right the version after modifications (V_{af}). A DBpedia user would like to track the entities that are added between versions and specifically to know which are the added persons that work in academia. Each person may have several descriptive properties, like name, birthDate and deathDate, and may be further enriched with descriptive properties related to professional affairs, like employer, title, activeYearsStartYear and activeYearsEndYear. In Figure 1 right, the addition of "Margery Claire Carlson" entity along with its descriptive properties is depicted. It is an entity of type person, with one employer, the "Northwestern University", which is of type university. Computing the diff between these two versions, as a set of added/deleted triples, totally misses capturing change semantics, as well as possible dependencies among changes. Instead, Figure 2 depicts an intuitive and descriptive representation of how data changed, with focus on the user's interest and needs. Each node represents a change instance detected between the aforementioned versions. Change instances on leaf nodes (in grey) are fine-

[1] https://www.dbpedia.org/

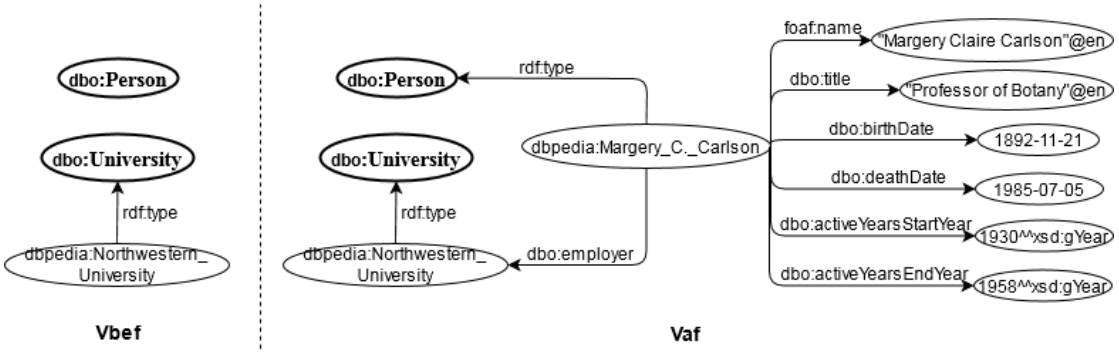


Figure 1 Sample part of DBpedia ontology, initial version (V_{bef}) and version after modifications (V_{af}).

grained and application/data-agnostic. Each one corresponds to an added or deleted triple and has a suitable name and descriptive parameters. They are simple change instances. The rest change instances (in white) are coarse-grained and application/data-specific, demonstrating structure and semantics suitable to the specific scenario of tracking persons that work in academia. The hierarchical structure indicates that a change instance is on top of others, demonstrating relations and dependencies among changes. They are complex change instances.

Consider the change instances Add_Person and Add_Name in Figure 2. They are specializations of the application/data-agnostic Add_Type_To_Individual and Add_Property_Instance respectively. The same holds for all similar change instances regarding descriptive properties. Add_Person_with_Details instance contains Add_Person, Add_Name, Add_BirthDate and Add_DeathDate instances, grouping the added person entity with a number of added descriptive properties with personal information. Note that, in general, Add_BirthDate and Add_DeathDate may not always be present, since they represent information that may be missing or in case of death date inappropriate. Add_Professional builds on top of Add_Person_with_Details, as further descriptive properties with professional life related information are added. Therefore, Add_Professional instance is a specialization of Add_Person_with_Details, where the added person (dbpedia:Margery_C._Carlson) has at least one employer (dbpedia:Northwestern_University). The change instances Add_Title, Add_ActiveYearsStartYear and Add_ActiveYearsEndYear represent secondary changes that may happen when adding a professional. Finally, Add_Academic_Professional further specializes Add_Professional and is defined on top of it, modeling the case where all the employers of the added professional are universities.

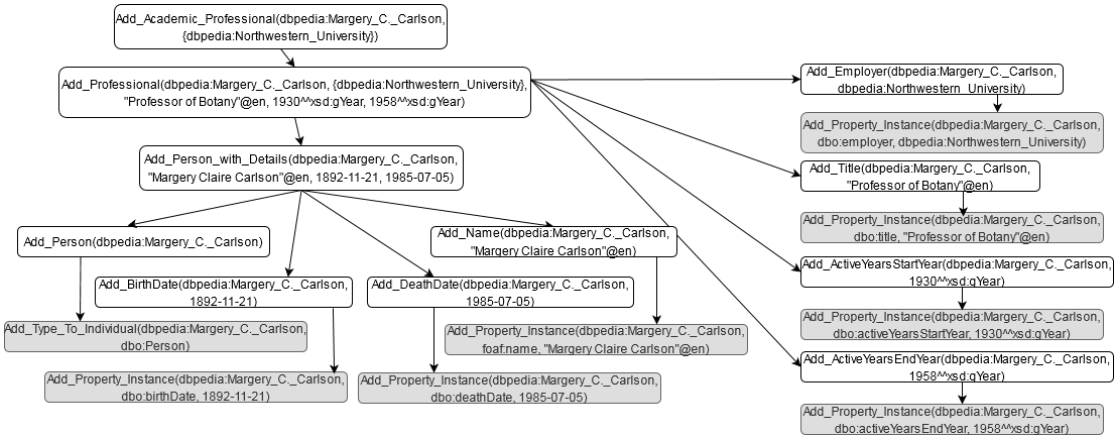


Figure 2 Hierarchy of detected simple and complex change instances (in grey and white fill respectively) for the sample part of DBpedia ontology presented in Figure 1.

A DBpedia user may spend a lot of time and effort querying the respective diff and processing results, attempting to approximate the representation of Figure 2 and conclude what changed and how. Even if he is equipped with a set of predefined human-readable changes and a relevant detection algorithm, changes like Add_Academic_Professional that capture specific evolution scenarios could not be recorded. Instead, assuming a two-level representation of changes via simple and complex changes would balance the needs. On the one hand, simple changes offer a first layer of primitive changes following the RDF data model. Each added or deleted triple is mapped to a specific simple change. On the other hand, complex changes offer a second layer of user-defined changes, grouping other changes into logical units. In this way, application/data-specific changes, dependencies between them and multiple interpretations of evolution can be captured. Additionally, a dedicated intuitive language for defining complex changes would facilitate the process. A complex change can be uniquely identified via a name, can be described by a set of parameters and defined as a pattern comprised of simple changes, other already defined complex changes and constraints guiding which change instances are grouped in order to form a new complex change instance. Complex change definitions may constitute a registry of reusable components/patterns that can be used for defining new dependent changes. Finally, the representation of the detected complex change instances as RDF data allows querying the relations and dependencies among changes via SPARQL [31] and the demonstration of change hierarchy by any graph visualization tool for RDF data (Antoniazzi and Viola (2018) [2]).

3.3. Simple and Complex changes on RDF(S) Knowledge Bases

Modeling changes as first class citizens involves taking into account granularity and semantics of changes. Granularity poses the question of having fine-grained or coarse-grained changes. Fine-grained changes have the advantage of describing primitive changes, while coarse-grained changes provide semantics and conciseness by grouping primitive changes in logical units. Semantics poses the question of having application/data-agnostic or -specific changes. Application/data-agnostic changes describe modifications that appear in a specific model, constituting a fixed set of generic changes. Application/data-specific changes suit specific use-cases and may be user-defined, allowing multiple interpretations of evolution.

As a result, changes are distinguished between simple and complex changes. Simple changes constitute a fixed set of fine-grained, application/data-agnostic changes. Complex changes are coarse-grained, user-defined, application/data-specific changes providing richer semantics on how data changed. This section provides definitions regarding simple and complex changes.

Definition 1: A simple change s is a tuple (n, P), where:

- n is the name of s, which must be unique.
- P is the list of descriptive parameters of s, where each one has a unique name within s.

For simple changes we rely on Papavasileiou et al. (2013) [45]. Annex I summarizes the simple changes considered. They are additions, deletions and terminological changes (rename, split, merge) of RDF(S) entities (classes, properties, individuals).

Simple changes verify completeness and unambiguity properties, constituting a first layer of human-readable changes. These properties were introduced in Papavasileiou et al. (2013) [45] and guarantee that simple change detection process exhibits a sound and deterministic behavior. Simple change detection is performed over a layer of "low-level" changes constituted by triple additions and deletions among dataset versions. Essentially, what is needed to be guaranteed is that each change that a dataset underwent is properly captured by one, and only one, simple change. Thus, low-level changes are "assigned" to simple changes, so that they are partitioned into simple changes. Completeness and unambiguity dictate that this partitioning is perfect. In a nutshell: Completeness guarantees that all low-level changes are associated with at least one simple change, making the reported delta complete (not

missing any change). Unambiguity guarantees that no race conditions emerge between simple changes attempting to be detected over the same low-level change. The combination of these properties guarantees that the delta is produced in a complete and deterministic manner. Further details on completeness and unambiguity can be found in Papavasileiou et al. (2013) [45] and Roussakis et al. (2015) [53].

As already stated, simple changes are fine-grained, i.e. they cannot be decomposed in more granular changes. This holds for additions and deletions, but not for terminological changes, as they can be expressed as additions and deletions plus extra conditions. For example, a class rename can be considered as an add class plus a delete class, which have the same "neighborhood" (properties, connections to classes). However, they are preferred to be simple changes in order to distinguish at simple change level real additions or deletions from virtual ones representing terminological changes. Thus, simple changes' set is not minimal.

Definition 2: A complex change c is a quadruple (n, P, D, F), where:

- n is the name of c, which must be unique and different from the simple change names.
- P is the list of descriptive parameters of c, where each one has a unique name within c.
- D is the set of simple (D_S) and complex changes (D_C) that c comprises of, where $D = D_C \cup D_S$, $D_C \cap D_S = \emptyset$ and $D \neq \emptyset$.
- F is the set of constraints (F_C) that changes in D verify and bindings (F_B) specifying the parameters in P, where $F = F_C \cup F_B$ and $F_C \cap F_B = \emptyset$. Constraints are on changes (F_C^{car}) or change parameters (F_C^{par}), where $F_C = F_C^{car} \cup F_C^{par}$ and $F_C^{car} \cap F_C^{par} = \emptyset$.

A complex change is defined in terms of simple or other complex changes verifying constraints. Constraints specialize its meaning and are divided into those defined on changes and those on change parameters. Bindings specify complex change parameter values. Section 3.4 includes the details regarding the specific types of constraints and bindings.

Note that Definitions 1 and 2 actually define a *class* of simple and complex changes respectively and not the concrete changes. The terms simple change and complex change will be used as a shorthand for any concrete simple/complex change in the respective class.

The ultimate goal of supporting simple and complex changes is detecting actual instances between dataset versions. The detection process leads into instantiating change parameters with values, indicating that specific data elements are affected by a change. Definitions 3 and

4 define simple and complex change instances. Figure 2 presents simple and complex change instance examples.

Definition 3: A simple change instance of a simple change (n, P), is a tuple (n, V) where V is an instantiation of the parameters P.

Definition 4: A complex change instance of a complex change (n, P, D, F), is a tuple (n, V) where V is an instantiation of the parameters P.

For simple change detection we rely on Papavasileiou et al. (2013) [45]. For complex change detection an appropriate algorithm is presented in Section 3.5. Definition 5 defines when a complex change instance is detected. Definitions 6 and 7 define possible relations among change instances, reflecting the relations and dependencies between changes.

Definition 5: Let $c = (n, P, D, F)$ be a complex change and V_{bef}, V_{af} two dataset versions. A complex change instance $c_i = (n, V)$ is detected if: (1) for all changes in D instances are detected between V_{bef} and V_{af}, forming D_i, such that constraints in F_C are verified on D_i, V_{bef} and V_{af}, (2) bindings in F_B are applied on D_i forming V, and (3) D_i is maximal.

The set of change instances D_i corresponding to c_i *verifies* the complex change c.

Definition 6: Let c_i be an instance of complex change c and D_i the corresponding set of change instances verifying c. c_i *contains* the change instances in D_i.

Containment property is transitive: if complex change instance c_i contains complex change instance c_j and c_j contains change instance c_k in turn, then it also holds that c_i contains c_k.

Definition 7: Let c_i and c_i' be two complex change instances, where c_i does not contain c_i' and vice versa. They are *overlapping* if they both contain at least one common simple or complex change instance.

Overall, complex change instances may form a hierarchy due to containment and overlaps. As an example consider complex change instances in Figure 2.

Note that the dynamics model followed, detecting changes between versions, propagates some limitations to our approach. First, the order in which changes actually occurred cannot

be captured, since version-based approaches are agnostic of time between versions. Additionally, if one triple is deleted and then added back between two dataset versions, then this change cannot be traced, since change detection identifies the triple in both versions. In the same context, complex change detection is performed between two successive versions; thus complex changes spanning across multiple successive versions are not captured. In these cases, following a careful and guided version issuing policy would minimize change loss. Finally, in this work we ignore blank nodes that can be avoided when data are published according to the *linked open data* paradigm.

3.4. A Language for Defining Complex Changes

This section presents an intuitive, user-friendly language based on change semantics for defining complex changes. Complex change definitions are used for detecting respective instances among dataset versions. In Section 3.4.1 the language syntax is provided, by means of EBNF specification, as well as details regarding the supported concepts. In Section 3.4.2 the language semantics are formally defined. In Section 3.4.3 a number of examples are discussed.

3.4.1. Syntax

Table 1 presents the EBNF specification of the proposed language.

Complex change definition, heading and body. A complex change definition is composed by a heading and a body. The heading contains a unique name and a list of descriptive parameters. The body contains the change list, as well as optionally the filter list and the binding list. The change list defines the changes that the complex change comprises of, as well as the cardinality each one may have. The filter list defines filter expressions with constraints that the changes in the change list should verify. The binding list defines how complex change descriptive parameters are evaluated. A complex change definition is *nested* if complex changes appear in its change list.

Parameters. Change parameters may be categorized based on two criteria: (1) the type of values they may evaluate, (2) whether they may evaluate into empty value.

Based on the first criterion, parameters are distinguished into those that evaluate into type set and those that evaluate into scalar values. In order to distinguish these parameter types,

parameters evaluating into scalar values start with a lowercase letter, while those evaluating into sets with an uppercase letter.

Based on the second criterion, parameters are distinguished into those that may evaluate into an empty value and those that always should have a non-empty value. In order to distinguish these parameter types, parameters that may evaluate into an empty value have an "OPT" suffix (denoting optional). Also, they are referred as *optional*. Thus, a complex change may be defined to be tolerant in partially performed changes in lower levels in change hierarchy.

Cardinality constraints. They determine whether a change in the change list may group multiple instances. Specifically, they determine that there might be zero, one or multiple instances of a specific change to be contained into respective complex change instances. The default cardinality is one. Therefore, when no notation is defined, cardinality one is inferred. Also, the following notations hold: "+" for at least one change instance, "?" for zero or one, "*" for zero or more. Note that cardinality constraints are constraints on changes.

A change (in the change list) is *mandatory* in case of cardinality one or "+". A change is *optional* in case of cardinality "?" or "*". In case of an optional change, if no instance is detected, the respective complex change can be still detected. Thus, a complex change may be defined to be flexible and tolerant in partially performed modifications of minor significance.

Filter expressions. They determine constraints that the parameters of the changes included in the change list should verify. They are distinguished into *primitive* and *composite*.

Primitive filter expressions cannot break down into simpler ones. There are four types of constraints that form primitive filter expressions: (1) testing value constraints, (2) relational constraints, (3) pre/post-conditions, (4) functions. Also, these types of constraints may be augmented via quantified expressions. Primitive filter expressions are also distinguished into unary and binary, based on whether they involve one or two change parameters. It is worth noting that in terms of this work, a parameter that may evaluate into an empty value may be involved only on unary filters. While the nature of binary filters is to interconnect changes, if they involve such parameters the connection among changes will not be always established (if the parameter evaluates into an empty value). This contradicts the binary filter's goal.

Composite filter expressions are formed when combining primitive filter expressions via boolean operators. Specifically, logical AND, OR, NOT may be used.

Table 1 The EBNF specification of the complex change definition language

Complex Change Definition, Heading, Body, Parameters
1 ***complex-change-definition =*** 'CREATE COMPLEX CHANGE' heading '{' body '}'';' ;
2 ***heading =*** name '(' parameter-list ')' ;
3 ***parameter-list =*** identifier ***{***', ' identifier***}*** ;
4 ***body =*** change-list ***[***'; ' filter-list***]*** ***[***'; ' binding-list***]*** ;
5 ***name =*** STRING ;
6 ***identifier =*** id-scalar **|** id-set ;
7 ***id-scalar =*** id-scalar-nonempty **|** id-scalar-empty ;
8 ***id-set =*** id-set-nonempty **|** id-set-empty ;
9 ***id-scalar-empty =*** LOWERCASE_LETTER ***{***LETTER**|**DIGIT***}*** 'OPT' ;
10 ***id-set-empty =*** CAPITAL_LETTER ***{***LETTER**|**DIGIT***}*** 'OPT' ;
11 ***id-scalar-nonempty =*** LOWERCASE_LETTER ***{***LETTER**|**DIGIT***}*** ;
12 ***id-set-nonempty =*** CAPITAL_LETTER ***{***LETTER**|**DIGIT***}*** ;
Change List, Cardinalities
13 ***change-list =*** 'CHANGE LIST' change ***{***', ' change***}*** ;
14 ***change =*** change-heading ***[***cardinality***]*** ;
15 ***change-heading =*** change-name '(' parameter-list ')' ;
16 ***change-name =*** name **|** NAMES OF SUPPORTED SIMPLE CHANGES ;
17 ***cardinality =*** '+'**|**'?'**|**'*' ;
Filter List
18 ***filter-list =*** 'FILTER LIST' or-filter-expr ***{***', ' or-filter-expr***}*** ;
19 ***or-filter-expr =*** and-filter-expr ***{***'||' and-filter-expr***}*** ;
20 ***and-filter-expr =*** neg-filter-expr ***{***'&&' neg-filter-expr***}*** ;
21 ***neg-filter-expr = [***'!'***]*** filter-expr ;
22 ***filter-expr =*** bracketed-expr **|** expr ;
23 ***bracketed-expr =*** '(' or-filter-expr ')' ;
24 ***expr =*** unary-expr **|** binary-expr ;
25 ***unary-expr = [***quantification-1***]*** unary-constr ;
26 ***binary-expr = [***quantification-2***]*** binary-constr ;
27 ***quantification-1 =*** 'for' ***(***'each'**|**'some'**|**'none'***)*** id-scalar-nonempty 'in' id-set ':';
28 ***unary-constr =*** test-val-constr **|** pre-post-constr-1 **|** fun-constr-1;
29 ***test-val-constr =*** test-val-scalar-1 **|** test-val-scalar-2 **|** test-val-set ;
30 ***test-val-scalar-1 =*** id-scalar bin-op-scalar-1 value ;
31 ***test-val-scalar-2 =*** id-scalar bin-op-scalar-2 set ;
32 ***test-val-set =*** id-set bin-op-set set ;
33 ***pre-post-constr-1 = ((***'(' id-scalar ', ' URI ', ' value ')'***)*** **|** ***(***'(' URI ', ' id-scalar ', ' value ')'***)*** **|** ***(***'(' URI ', ' URI ', ' id-scalar ')'***)) [***'inferred'***]*** ***(***'in' **|** 'not in'***)*** ***(***'Vbef' **|** 'Vaf'***)*** ;
34 ***fun-constr-1 =*** name '(' ***(*** identifier **|** ***(***identifier ', ' constant***)*** **|** ***(***constant ', ' identifier***))*** ')' ;
35 ***quantification-2 =*** 'for' ***(***'each'**|**'some'**|**'none'***)*** id-scalar-nonempty 'in' id-set-nonempty ':' ***[***'for' ***(***'each'**|**'some'**|**'none'***)*** id-scalar-nonempty 'in' id-set-nonempty ':'***];***
36 ***binary-constr =*** rel-constr **|** pre-post-constr-2 **|** fun-constr-2 ;
37 ***rel-constr =*** rel-scalar-1 **|** rel-scalar-2 **|** rel-set ;
38 ***rel-scalar-1 =*** id-scalar-nonempty bin-op-scalar-1 id-scalar-nonempty;
39 ***rel-scalar-2 =*** id-scalar-nonempty bin-op-scalar-2 id-set-nonempty;
40 ***rel-set =*** id-set-nonempty bin-op-set id-set-nonempty ;
41 ***pre-post-constr-2 = ((***'(' id-scalar-nonempty ', ' URI ', ' id-scalar-nonempty ')'***)*** **|** ***(***'(' id-scalar-nonempty ', ' id-scalar-nonempty ', ' value ')'***)*** **|** ***(***'(' URI ', ' id-scalar-nonempty ', ' id-scalar-nonempty ')'***)) [***'inferred'***]*** ***(***'in' **|** 'not in'***)*** ***(***'Vbef' **|** 'Vaf'***)*** ;
42 ***fun-constr-2 =*** name '(' ***(***id-scalar-nonempty **|** id-set-nonempty***)*** ', ' ***(***id-scalar-nonempty **|** id-set-nonempty***)*** ')' ;

```
43 bin-op-scalar-1 = '=' | '!=' | '>' | '<' | '>=' | '<=' ;
44 bin-op-scalar-2 = 'in' | 'not in' ;
45 bin-op-set = '=' | '!=' | 'subSet' | 'properSubset' | 'superSet'
|'properSuperset';
46 constant = set | value ;
47 set = '{' value-list '}' ;
48 value-list = value {', ' value} ;
49 value = URI | LITERAL ;
```

Binding List

```
50 binding-list = 'BINDING LIST' binding {', ' binding} ;
51 binding = (id-scalar 'as' id-scalar)|(id-set 'as' id-
set)|(aggregate 'as' id-set);
52 aggregate = 'union(' identifier ')' ;
```

Testing value constraints. Testing value constraints limit a parameter value against a given constant. They are actually unary filters. Their form is presented in Table 1, lines 29-32. The supported binary operators are the typical ones, as presented in Table 1, lines 43-45. They may be on scalar parameters ($=, !=, >, <, >=, <=$), on set parameters ($=, !=, \supset$ as $properSuperset$, $\subset$ as $properSubset$, $\supseteq$ as $superSet$, $\subseteq$ as $subSet$), or may involve both scalar and set parameters ($\in$ as in, $\notin$ as $not\ in$).

Relational constraints. Relational constraints involve two change parameters defining how changes are connected. They are actually binary filters. Their form is presented in Table 1, lines 37-40. The supported binary operators are the same with the ones used in testing value constraints, as presented in Table 1, lines 43-45.

Pre-/Post-conditions. Pre-/post-conditions define how parameters are related in the version before (V_{bef}) or after (V_{af}) the change. They state whether a triple must or must not exist in the version before or after and whether a triple may be inferred. In case of inference, a flag "inferred" is used. These constraints may be unary or binary depending on the number of change parameters they involve. Their form is presented in Table 1, lines 33 and 41.

Functions. Function constraints involve predefined functions of boolean return type. As an example, consider common functions on strings, like checking whether a string contains another given string. These constraints may be unary or binary depending on the number of change parameters they involve. Their form is presented in Table 1, lines 34 and 42.

Quantified expressions. Quantified expressions allow to write conditions on elements of set parameters. Thus, quantification augments primitive filters on scalar parameters so they evaluate into elements of set parameters. They may have one of the following forms: (1) $\{\forall, \exists, \nexists\}\, x \in X : f(x)$, (2) $\{\forall, \exists, \nexists\}\, x \in X : f(x, y)$, (3) $\{\forall, \exists, \nexists\}\, x \in X : \{\forall, \exists, \nexists\}\, y \in$

$Y: f(x, y)$, where $f(x)$ and $f(x, y)$ are primitive constraints on parameters evaluating into scalar values. Their syntax is presented in Table 1, lines 25-27 and 35.

Bindings. Parameter bindings determine how complex change descriptive parameters are evaluated. In its simplest form, a binding assigns the identifier on the left to the identifier on the right, via operator *as*. In this way, it is defined that the identifier on the right, which represents a complex change parameter, equals the identifier on the left, which represents the parameter value of a change in its change list. This type of binding may be omitted and thus inferred by repeating each descriptive parameter into the respective contained changes and constraints. Moreover, a binding assigns the result of the aggregate function *union* over an identifier to another identifier (on the right), via operator *as*. In this way, it is defined that the second identifier, which represents a complex change parameter, equals the union of the parameter values of a change with cardinality "+" or "*", whose parameter is passed as an argument in the aggregate function *union*. Obviously, the complex change parameters that are evaluated with the latter form are of type set. A binding that involves the union aggregate function is useful in case of changes with cardinality "+" or "*". The syntax is presented in Table 1, lines 50-52.

3.4.2. Semantics

The proposed language is essentially a pattern-matching language. The body of a complex change definition constitutes a *change pattern expression* (or a *change pattern*), while the head indicates how to construct a complex change instance. The body is matched against a set of change instances I, between two dataset versions, V_{bef} and V_{af}, to obtain a set of bindings for the variables in the body, and then based on the head these bindings are used to produce the actual change instance. Note that, we say that the generated complex change instance contains the change instances in I that correspond to the respective bindings. The semantics definition is influenced by SPARQL semantics definition as in Perez, Arenas and Gutierrez (2009) [46] and Kaminski, Kostylev and Cuenca Grau (2017) [32], but adapted to our language needs.

A change (simple or complex) is a tuple (n, V_r^n), where n is the change name and V_r^n is a list of variables (scalar or set), out of which one is a *change variable* and the rest are *descriptive variables*. With respect to Definitions 1 and 2, the change variable represents the change identifier s and c respectively, the descriptive variables represent the change parameters P, while for complex changes the *heading* is considered. A change instance (simple or complex)

is a tuple (n, V_l^n), where n is the change name and V_l^n is a list of values (scalar or set) serving as instantiations of the respective variables. Additionally, consider the existence of an infinite set V_l of possible values, scalar or set, $(V_l = V_l^{scalar} \cup V_l^{set})$ and an infinite set V_r of variables, scalar or set, including optional variables, $(V_r = V_r^{scalar} \cup V_r^{set})$ disjoint from V_l.

3.4.2.1. Baseline Algebra and Semantics

Change Pattern Expression. While the formal syntax of the proposed language was presented in Section 3.4.1, in order to define the semantics an algebraic formalization is followed. The binary operators AND, OPT (i.e. optional) and FILTER are used. AND is used for concatenating mandatory changes, i.e. those with cardinality 1 or "+", while OPT is used for optional changes, i.e. those with cardinality "?" or "*", in both cases instead of comma symbol (",") in formal syntax. FILTER is used for filter expressions. A change pattern expression is defined recursively as follows:

(1) A change (n, V_r^n), where $V_r^n \subset V_r$, is a change pattern (*primitive change pattern*).

(2) If P is a primitive change pattern and R is a built-in filter expression, then the expression $(P\ FILTER\ R)$ is a change pattern (*filter primitive change pattern*).

(3) If P_1 and P_2 are change patterns, then the expression $(P_1\ AND\ P_2)$ is a change pattern (*conjunction change pattern*).

(4) If P_1 is a change pattern and P_2 is a primitive change pattern or a filter primitive change pattern, then the expression $(P_1\ OPT\ P_2)$ is a change pattern (*optional change pattern*). If P_1 is a change pattern and P_2 is an optional change pattern $P_2 = (P_A\ OPT\ P_B)$ where P_A and P_B are primitive change patterns or filter primitive change patterns or optional change patterns similar to P_2, then $(P_1\ OPT\ P_2)$ is a change pattern (*optional change pattern*).

(5) If P is a change pattern and R is a built-in filter expression, then the expression $(P\ FILTER\ R)$ is a change pattern (*filter change pattern*).

In case 4, nested optional change patterns may be defined based on primitive change patterns and filter primitive change patterns. In this case an optional change is *dependent* to another optional change, which in turn is dependent to another, and so on, ending up to a mandatory change which is part of a change pattern. This pattern of optional changes is named *optional change path*. Overall, it is not meaningful to define a complex change comprising of optional changes only and each optional change is meaningful in the context of another change.

In cases 2 and 5, a built-in filter expression is constructed using elements of the set $V_l \cup V_r$, logical connectives $(\neg,\wedge,\vee)$, several symbols and constraints as described in Section 3.4.1, evaluating into a boolean value. Formally, the built-in filter expressions below are considered:
(1) If $x, y \in V_r^{scalar}$, $X, Y \in V_r^{set}$, $v \in V_l^{scalar}$, $V \in V_l^{set}$ then the following are built-in filter expressions:
- *Equality symbol* $(=)$. $x = v$, $X = V$, $x = y$, $X = Y$ are built-in filter expressions.
- *Inequality symbols* $(>, <, >=, <=, \supset, \subset, \supseteq, \subseteq, !=)$. $x > v$, $X \supset V$, $x > y$, $X \supset Y$ are built-in filter expressions, and similar holds for the rest of the symbols.
- *Existential symbols* $(\in, \notin)$. $x \in V$, $x \notin V$, $x \in Y$, $x \notin Y$ are built-in filter expressions.
- *Pre-/post-conditions.* $t \in V_{bef}$, $t \notin V_{bef}$, $t \in V_{af}$, $t \notin V_{af}$ are built-in filter expressions, where t is a tuple from $\{(\{v \in V_l^{scalar} | isIRI(v)\} \cup V_r^{scalar}) \times (\{v \in V_l^{scalar} | isIRI(v)\} \cup V_r^{scalar}) \times (V_l^{scalar} \cup V_r^{scalar})\} - \{V_r^{scalar} \times V_r^{scalar} \times V_r^{scalar}\}$. Similar expressions are defined while taking into consideration the inferred tuples in V_{bef} and V_{af} as well. In these cases, $Inf(V_{bef})$ and $Inf(V_{af})$ datasets include explicitly those tuples.
- *Functions.* $fun(x)$, $fun(x, v)$, $fun(v, x)$ and $fun(x, y)$, with return type boolean are built-in filter expressions. Similar expressions are defined with variables of type set and set values.
- *Quantified expressions.* $\forall x \in X: f(x)$, $\exists x \in X: f(x)$, $\nexists x \in X: f(x)$, $\forall x \in X: f(x, y)$, $\exists x \in X: f(x, y)$, $\nexists x \in X: f(x, y)$, $\forall x \in X: \forall y \in Y: f(x, y)$, $\forall x \in X: \exists y \in Y: f(x, y)$, $\exists x \in X: \exists y \in Y: f(x, y)$, $\nexists x \in X: \nexists y \in Y: f(x, y)$ are built-in filter expressions, where $f(x)$ and $f(x, y)$ may be any of the aforementioned built-in expressions on parameters evaluating into scalar values.
Note that, an optional variable $(xOPT, XOPT)$ may be involved only on unary filters.
(2) If R_1 and R_2 are built-in filter expressions, then $(\neg R_1)$, $(R_1 \vee R_2)$ and $(R_1 \wedge R_2)$ are built-in filter expressions.

Mappings and Set of Mappings. Let P be a change pattern expression, R be a built-in filter expression and t be a tuple. $var(P)$ denotes the set of variables occuring in P, $var(R)$ in R, and $var(t)$ in t.

In order to define the semantics, the following terminology has to be introduced. A mapping μ_c from V_r to V_l is a partial function $\mu_c : V_r \rightarrow V_l$. Abusing the notation, for a primitive change pattern c we denote by $\mu_c(c)$ the change instance obtained by replacing the variables in c according to μ_c. The domain of μ_c, denoted by $dom(\mu_c)$, is a subset of V_r where μ_c is defined. Two mappings μ_{c1} and μ_{c2} are *compatible* when for all $x \in dom(\mu_{c1}) \cap dom(\mu_{c2})$, it is the case that $\mu_{c1}(x) = \mu_{c2}(x)$, i.e. when $\mu_{c1} \cup \mu_{c2}$ is also a mapping. Intuitively, μ_{c1} and μ_{c2} are compatible if μ_{c1} can be *extended* with μ_{c2} to obtain a new mapping, and vice versa.

Two mappings with disjoint domains are always compatible, and empty mapping $\mu_{c\emptyset}$ (i.e. the mapping with empty domain) is compatible with any other mapping.

Notice that μ_c is defined over V_r and its set of destination is V_l, which include variables and values of both scalar and set type respectively. This allows variables of type scalar/set to evaluate into a scalar/set value. Furthermore, in case of optional variables, they may also evaluate into an empty value ($\emptyset$). Finally, μ_c allows the evaluation of both change and descriptive variables.

Let Ω_1 and Ω_2 be sets of mappings. The join of, union of and difference between Ω_1 and Ω_2 are defined bellow. Based on these operators, the left-outer join is defined.

$\Omega_1 \bowtie \Omega_2 = \{\mu_{c1} \cup \mu_{c2} | \mu_{c1} \in \Omega_1, \mu_{c2} \in \Omega_2 \text{ and } \mu_{c1}, \mu_{c2} \; are \; compatible \; mappings\}$

$\Omega_1 \cup \Omega_2 = \{\mu | \mu \in \Omega_1 \text{ or } \mu \in \Omega_2\}$

$\Omega_1 \setminus \Omega_2 = \{\mu \in \Omega_1 | \text{ for all } \mu' \in \Omega_2, \mu \; and \; \mu' \; are \; not \; compatible \; mappings\}$

$\Omega_1 \rtimes \Omega_2 = (\Omega_1 \bowtie \Omega_2) \cup (\Omega_1 \setminus \Omega_2)$

Semantics Definition. Based on the above, the semantics of change pattern expressions can be defined as a function $[\![\cdot]\!]_I$ which takes as input a change pattern expression and returns a set of mappings.

Definition 8. The *evaluation* of a change pattern P over a set of change instances I, denoted by $[\![P]\!]_I$, is defined recursively as follows.

(1) If P is a primitive change pattern c, then $[\![P]\!]_I = \{\mu_c | dom(\mu_c) = var(c) \; and \; \mu_c(c) \in I\}$.

(2) If P is $(P_1 \; AND \; P_2)$, then $[\![P]\!]_I = [\![P_1]\!]_I \bowtie [\![P_2]\!]_I$.

(3) If P is $(P_1 \; OPT \; P_2)$, then $[\![P]\!]_I = [\![P_1]\!]_I \rtimes [\![P_2]\!]_I$.

The semantics of filter expressions goes as follows. Given a mapping μ_c and a built-in filter expression R, μ_c satisfies R denoted by $\mu_c \vDash R$, if:

(1) R is $x = v$, $x \in dom(\mu_c)$ and $\mu_c(x) = v$.

(2) R is $X = V$, $X \in dom(\mu_c)$ and $\mu_c(X) = V$.

(3) R is $x = y$, $x \in dom(\mu_c)$, $y \in dom(\mu_c)$ and $\mu_c(x) = \mu_c(y)$.

(4) R is $X = Y$, $X \in dom(\mu_c)$, $Y \in dom(\mu_c)$ and $\mu_c(X) = \mu_c(Y)$.

For the inequality symbols $(>, <, >=, <=, \supset, \subset, \supseteq, \subseteq, !=)$ similar definitions hold.

(5) R is $x \in V$, $x \in dom(\mu_c)$ and $\mu_c(x) \in V$.

(6) R is $x \in Y$, $x \in dom(\mu_c)$, $Y \in dom(\mu_c)$ and $\mu_c(x) \in \mu_c(Y)$.

For the rest of existential symbols $(\notin)$ similar definitions hold.

(7) R is $t \in V_{bef}$, $\forall x \in var(t) \quad x \in dom(\mu_c)$ and $t|_{\forall x \in var(t):\, \mu_c(x)} \in V_{bef}$, where $t|_{\forall x \in var(t):\, \mu_c(x)}$ represents the tuple t where each of its variables x is substituted by $\mu_c(x)$.

(8) R is $t \notin V_{bef}$, $\forall x \in var(t) \quad x \in dom(\mu_c)$ and $t|_{\forall x \in var(t):\, \mu_c(x)} \notin V_{bef}$, where $t|_{\forall x \in var(t):\, \mu_c(x)}$ represents the tuple t where each of its variables x is substituted by $\mu_c(x)$.

For the rest of pre-/post-conditions ($t \in V_{af}, t \notin V_{af}$ and with inference) similar definitions hold.

(9) R is $fun(x)$, $x \in dom(\mu_c)$ and $fun\big(\mu_c(x)\big)$ is true.

For the rest of functions ($fun(x, v)$, $fun(v, x)$, $fun(x, y)$, and with variables of type set and set values) similar definitions hold.

(10) R is $\forall x \in X: f(x)$, $X \in dom(\mu_c)$ and $\forall v \in \mu_c(X)\ f(v)$ is true.

(11) R is $\exists x \in X: f(x)$, $X \in dom(\mu_c)$ and $\exists v \in \mu_c(X)\ f(v)$ is true.

(12) R is $\nexists x \in X: f(x)$, $X \in dom(\mu_c)$ and $\forall v \in \mu_c(X)\ f(v)$ is false.

(13) R is $\forall x \in X: f(x, y)$, $X \in dom(\mu_c)$, $y \in dom(\mu_c)$ and $\forall v \in \mu_c(X)\ f\big(v, \mu_c(y)\big)$ is true.

(14) R is $\exists x \in X: f(x, y)$, $X \in dom(\mu_c)$, $y \in dom(\mu_c)$ and $\exists v \in \mu_c(X)\ f\big(v, \mu_c(y)\big)$ is true.

(15) R is $\nexists x \in X: f(x, y)$, $X \in dom(\mu_c)$, $y \in dom(\mu_c)$ and $\forall v \in \mu_c(X)\ f\big(v, \mu_c(y)\big)$ is false.

For the rest of quantified expressions ($\forall x \in X: \forall y \in Y: f(x, y)$, $\forall x \in X: \exists y \in Y: f(x, y)$, $\exists x \in X: \exists y \in Y: f(x, y)$, $\nexists x \in X: \nexists y \in Y: f(x, y)$) similar definitions hold.

(16) R is $f(xOPT)$, where f may be any of the aforementioned built-in unary expressions on a scalar variable, $xOPT \in dom(\mu_c)$ and $\mu_c(xOPT) = \emptyset$, or $xOPT \in dom(\mu_c)$ and $\mu_c(xOPT) \neq \emptyset$ and $f\big(\mu_c(xOPT)\big)$ is true. Similar holds for an optional variable of type set ($XOPT$).

(17) R is $(\neg R_1)$, R_1 is a built-in filter expression, and it is not the case that $\mu_c \vDash R_1$.

(18) R is $(R_1 \vee R_2)$, R_1 and R_2 are built-in filter expressions, and $\mu_c \vDash R_1$ or $\mu_c \vDash R_2$.

(19) R is $(R_1 \wedge R_2)$, R_1 and R_2 are built-in filter expressions, and $\mu_c \vDash R_1$ and $\mu_c \vDash R_2$.

Definition 9. Given a set of change instances I and a filter expression $(P\ FILTER\ R)$,

$$[\![(P\ FILTER\ R)]\!]_I = \{\mu_c \in [\![P]\!]_I | \mu_c \vDash R\}.$$

3.4.2.2. Extended Algebra and Semantics

In order to formally define the binding construct of a complex change definition and multiple cardinality ("+", "*"), the presented algebra has to be extended with assignment and aggregation constructs. For this, the semantics of a change pattern should depend on a set of change instances I as well as on a mapping μ_c, called *environment*. The semantics of the extended change pattern expressions can be defined as a function $[\cdot]_I^{\mu_c}$ which takes as input an

extended change pattern expression and returns a set of mappings. The *environment evaluation* $[P]_I^{\mu_c}$ of a change pattern P over a set of change instances I with respect to a mapping μ_c is defined the same as $[\![P]\!]_I$ when $\mu_c = \emptyset$. Therefore, for the change patterns defined in Section 3.4.2.1 $[\![P]\!]_I = [P]_I^{\emptyset}$.

Extend Operator. First, the *Extend* operator is presented, which captures the complex change assignment construct, providing the algebraic means of assigning an expression to a variable. Note that in terms of the proposed complex change language, the expression might be a variable or the result of an aggregation function. Therefore, the algebra is extended so that: $Extend(x, E, P)$ is a change pattern, where x is a variable not in $var(P)$, E is an expression and P is a change pattern.

Definition 10. The evaluation of a change pattern $Extend(x, E, P)$ given a set of change instances I and an environment v is defined as follows:

$$[Extend(x, E, P)]_I^v = \left\{\mu_c | \mu_c' \in [P]_I^v, \mu_c = \mu_c' \cup \left\{x \mapsto [E]_I^{\mu_c'}\right\}\right\}.$$

Intuitively, $Extend$ assigns to variable x the evaluation of E in each solution mapping of P and the set of change instances I.

Group and Aggregate. Next, aggregation in terms of complex change definitions is formalized. The notion of groups is introduced: a group induces a partitioning of a change pattern's solution mappings into equivalence classes, each of which is determined by a key obtained from the evaluation of a list of variables. The list of variables for a complex change comprises of: (1) the change variables of changes it consists of with cardinality 1 or "?", since one respective instance is considered, and (2) the descriptive variables that are used in assignments without aggregation and correspond (only) to changes with cardinality "+" or "*", since even if multiple instances are considered, all of them should have a common value on these variables. If the list of variables is empty, then one group is assumed with all change pattern's solution mappings.

Definition 11. A v_l*-list* is a list of values in V_l. The evaluation $\left[V_r^g\right]_I^{\mu_c}$ of a variable list $V_r^g = \langle v_{r1}, \dots, v_{rn}\rangle$ over a set of change instances I with respect to a mapping μ_c is the v_l-list $\langle [v_{r1}]_I^{\mu_c}, \dots, [v_{rn}]_I^{\mu_c}\rangle$.

Definition 12. A *group* is a construct $\Gamma = Group(V_r^g, P)$, where V_r^g is a list of variables and P a change pattern. The evaluation $[\![\Gamma]\!]_I$ of a group $\Gamma = Group(V_r^g, P)$ over a set of change instances I is a partial function from v_l-lists to sets of mappings, that is defined for all v_l-lists $Key = [V_r^g]_I^{\mu_c}$ with $\mu_c \in [\![P]\!]_I$ as follows:

$$[\![\Gamma]\!]_I(Key) = \{\mu_c | \mu_c \in [\![P]\!]_I, [V_r^g]_I^{\mu_c} = Key\}.$$

Notice that the evaluation of groups is not dependent on environments, while the evaluation of v_l-lists it is.

Similar to *aggregate functions* proposed in standard query languages, *union* aggregate function allows to compute a single value for each group of solution mappings. Specifically, it calculates a set value for each group of solution mappings, based on the evaluation of a specific variable v_r over the solution mappings of each group. Suppose Λ be a set of mappings of variable v_r to values in V_l, then: $union(\Lambda) = \cup_{\forall\{v_r \mapsto v_l\} \in \Lambda}\{v_l\}$.

The aggregate construct is defined below, as a construct which computes a set value for each group, by means of union aggregate function.

Definition 13. An *aggregate* is a construct of the form $A = Aggregate(v_r, union, \Gamma)$, where v_r is a variable, $union$ is an aggregate function and $\Gamma = Group(V_r^g, P)$ is a group. The evaluation $[\![A]\!]_I$ of an aggregate $A = Aggregate(v_r, union, \Gamma)$ over a set of change instances I is a partial function from v_l-lists to values such that for each Key in the domain of $[\![\Gamma]\!]_I$,

$$[\![A]\!]_I(Key) = union(\{[v_r]_I^{\mu_c} | \mu_c \in [\![\Gamma]\!]_I(Key)\}).$$

3.4.3. Illustrative Examples

Examples 1-5 present complex change definitions regarding the changes discussed in Figure 1 on part of the DBpedia ontology. Examples 6-8 further elaborate on concepts of the proposed language and are based on the DBpedia ontology as well.

Example 1. Add_Person models the case where a new individual of type person is added. It is a specialization of simple change Add_Type_To_Individual, where the type equals to dbo:Person via a testing value constraint over parameter *type*. No binding is defined, as it is inferred by repeating the complex change parameter as parameter on the change in change list. Besides Add_Property_Instance no cardinality is defined as cardinality one is inferred.

```
CREATE COMPLEX CHANGE Add_Person(id) {
CHANGE LIST Add_Type_To_Individual(id, type) ;
FILTER LIST type=dbo:Person ; } ;
```

Example 2. Add_Name models the case where a new name property with value *n* is assigned to a person *id*. It is a specialization of simple change Add_Property_Instance, where the property equals to foaf:name via a testing value constraint over parameter *prop*. Bindings and cardinality are as in example 1. Similar definitions can be defined for all properties on person.

```
CREATE COMPLEX CHANGE Add_Name(id, n) {
CHANGE LIST Add_Property_Instance(id, prop, n) ;
FILTER LIST prop=foaf:name ; } ;
```

Example 3. Add_Person_with_Details models the case where a new person *id* is added with a number of descriptive properties assigned. Properties birth date and death date are optional, specifically zero or one property instance may be assigned to each person as defined by "?", since this information may be missing or death date may not be appropriate.

```
CREATE COMPLEX CHANGE Add_Person_with_Details(id, n, bD, dD) {
CHANGE LIST Add_Person(id), Add_Name(id, n), Add_BirthDate(id, bD) ?,
Add_DeathDate(id, dD) ? ; } ;
```

Example 4. Add_Professional is a specialization of Add_Person_with_Details and thus it is defined on top. It models the case where an added person is assigned several properties related to its professional activity, like employers, title, the start year and end year when being active. Since multiple employers may appear, cardinality "+" is used besides Add_Employer. Title, start year and end year may be missing, and thus cardinality "?" is used besides relevant changes, indicating zero or one instance. Parameter *E* holds all employers that the added person is connected with. This is defined with a *union* aggregate function in the binding list.

```
CREATE COMPLEX CHANGE Add_Professional(id, E, t, sY, eY) {
CHANGE LIST Add_Person_with_Details(id, n, bD, dD), Add_Employer(id,
e) +, Add_Title(id, t) ?, Add_ActiveYearsStartYear(id, sY) ?,
Add_ActiveYearsEndYear(id, eY) ? ;
BINDING LIST union(e) as E ; } ;
```

Example 5. Add_Academic_Professional is a specialization of Add_Professional and thus it is defined on top. It models the case where the added professional works only in academia. This is defined by a post-condition constraint on *E* using quantification.

```
CREATE COMPLEX CHANGE Add_Academic_Professional(id, E) {
CHANGE LIST Add_Professional(id, E, t, sY, eY) ;
FILTER LIST for each e in E : (e,rdf:type,dbo:University) in Vaf ; }
;
```

Example 6. Add_Professionals_withCommon_Employers is built on top of Add_Professional. It identifies all the added professionals that have the same employers, i.e. groups all the

Add_Professional change instances with the same value in parameter *E*. This is denoted by cardinality "+" besides Add_Professional change, while at the same time complex change parameter *E* equals the respective Add_Professional change parameter. Also, *I* holds all the professionals with the same employers *E*. This is defined by the *union* aggregate in the binding list. Notice, that if for example parameter *t* was also a complex change parameter, then the grouping would be the professionals with same employers *E* and title *t*.

```
CREATE COMPLEX CHANGE Add_Professionals_withCommon_Employers(I, E) {
CHANGE LIST Add_Professional(id, E, t, sY, eY) + ;
BINDING LIST union(id) as I ; } ;
```

Example 7. Add_Organization_with_ChildOrganisations models the case where a new organization is added together with its child-organizations, which may be more than one (cardinality "+"). This change is used in example 8. Changes Add_Organization and Add_ChildOrganization are defined similar to Add_Person and Add_Name respectively.

```
CREATE COMPLEX CHANGE Add_Organisation_withChildOrganisations(id, C){
CHANGE LIST Add_Organisation(id), Add_ChildOrganisation(id, chId) + ;
BINDING LIST union(chId) as C ; } ;
```

Example 8. Add_Organisation_Hierarchy models the case where a new organization is added together with any child-organizations, which in turn may have their child-organizations, forming overall a hierarchical structure with four levels. The notion of optional change path may be used in order to model the addition of such hierarchies, where in some cases may be complete while in others partial, since elements lower in the hierarchy may not appear. Here, one Add_Organization_with_ChildOrganisations is defined as mandatory change and two more as optional changes with cardinality "*", since zero, one or more organizations with child-organizations may be added lower in the hierarchy. The relational constraints (operator *in*) are used in the filter list to define the dependencies and connections among changes.

```
CREATE COMPLEX CHANGE Add_Organisation_Hierachy(id1, L2, L3, L4) {
CHANGE LIST Add_Organisation_withChildOrganisations(id1, C2),
Add_Organisation_withChildOrganisations(id2, C3) *,
Add_Organisation_withChildOrganisations(id3, C4) * ;
FILTER LIST id2 in C2, id3 in C3 ;
BINDING LIST C2 as L2, union(C3) as L3, union(C4) as L4 ; } ;
```

3.5. Complex Change Detection

Complex change detection is the process of identifying complex change instances. It requires as input a set of simple change instances detected between two dataset versions (S_i), the dataset versions (before V_{bef} and after V_{af}) and the complex change definitions that will be evaluated for detecting respective instances (C). For implementing the proposed language, we translate it into an already implemented language. As this approach concerns RDF data, we

choose to rely on SPARQL, which provides similar capabilities to the proposed language. Accordingly, simple and complex change instances, as well as dataset versions are encoded as RDF(S) data. Section 3.5.1 presents the complex change detection algorithm, Section 3.5.2 how change instances are represented in RDF(S), Section 3.5.3 the translation process for generating SPARQL queries, Section 3.5.4 the change instance generation process and Section 3.5.5 the correctness of the proposed implementation with respect to the language semantics.

3.5.1. Algorithm

The presented complex change detection algorithm, Algorithm 1, involves two steps: the first step handles nested definitions, the second produces complex change instances.

Algorithm 1: Complex Change Detection
Input: A set of complex changes C, a dataset version before V_{bef} and after V_{af}, a set of simple change instances S_i
Output: A set of complex changes instances I of C
1 $I \leftarrow \{\}$;
2 queue $Q \leftarrow postOrderDfs(C)$;//complex changes sorted based on dependencies
3 **while** $!Q.isEmpty()$ **do**
4 $c \leftarrow Q.dequeue()$;
5 $query \leftarrow createQuery\big(D(c), F(c)\big)$;
6 $resultSet \leftarrow exec\big(query, S_i, I, V_{bef}, V_{af}\big)$;
7 $I_c \leftarrow createInstances\big(resultSet, F_C^{car}(c)\big)$;
8 $I \leftarrow I \cup I_c$; //report instances
9 **end while**
10 **return** I ;

As for the first step, suppose a complex change c whose definition is based on a set of complex changes ($D_C \neq \emptyset$). The detection of c instances depends on detecting the instances of each complex change in D_C and therefore follows their detection. Note that mutually dependent complex changes are not supported. In general, complex change definitions constitute a directed acyclic graph, where nodes represent changes and edges dependencies between them. An edge departing from a complex change c arrives at changes in D_C according to its definition. Thus, detection follows a post-order depth-first scheme on the induced dependency graph by complex change definitions. This is stated in line 2 of Algorithm 1. postOrderDfs function call runs over the set of complex changes C identifying the dependencies among changes, returning a queue Q of all changes in C, where the order of elements defines the order in which they have to be detected.

As for the second step, for each complex change c in Q, instances are computed (lines 3-10). The main idea is that each complex change definition is translated into a SPARQL query plus a post process for computing respective complex change instances based on the query result. Simple and complex change instances as well as dataset versions are encoded as RDF(S) data, so that the constructed SPARQL queries are applied on them. Therefore, for each complex change an appropriate SPARQL query is created via createQuery function call (line 5). The query is executed on the detected change instances and dataset versions (line 6), in order to select change instances that verify the defined constraints. The query results are further elaborated through createInstances function call (line 7), so that selected changes are grouped based on cardinality. Computed instances are added into the set of instances to be reported I (line 8, initialized in line 1) and become available for the detection of dependent complex changes. Finally, the set of detected complex change instances I is returned (line 10).

3.5.2. RDF(S) Change Representation

The proposed schema describes the specification of each change and the instances under this schema describe the detected change instances. It actually forms a change vocabulary, with a dedicated namespace <http://dblab.ece.ntua.gr/change#> and prefix <ch>. The classes, properties and individuals are described below in detail.

Classes. A class for simple changes (ch:Simple_Change) and a class for complex changes (ch:Complex_Change) is used. Both are subclasses of a generic class for all changes (ch:Change). Also, for each simple change defined in Annex I a respective class is defined: ch:Add_Type_To_Individual, ch:Add_Property_Instance, etc. In total there are 38 classes for the simple changes, which are all subclasses of ch:Simple_Change class. Similarly, for each defined complex change in a set of complex changes C a class is defined, following the naming pattern <namespace>:<complex change name>, where a data-specific namespace is considered in line with the application domain of the complex changes. These classes are all subclasses of ch:Complex_Change class. For example, dbo:Add_Academic_Professional is a class for one of the complex changes defined in the running example.

Properties. For each descriptive parameter of a simple change, a property is considered and is named based on the simple change name and its index in the descriptive parameter list. The naming pattern is ch:<simple change name>_p<parameter index>. For each property, the domain is the respective simple change class and the range matches the value type it represents in the definition. For example, the simple change Add_Type_To_Individual(a, b)

has two properties defined: the first is ch:Add_Type_To_Individual_p1 with domain ch:Add_Type_To_Individual and range rdfs:Resource, and the second is ch:Add_Type_To_Individual_p2 with the same domain and range rdfs:Class. Similarly, for each descriptive parameter of a complex change a property is defined, where the naming pattern, domain and range are defined alike the simple changes.

Additionally, the property ch:contains is employed for modeling explicitly the containment relationship between change instances. The domain is ch:Complex_Change and the range is ch:Change, as only complex change instances may contain simple/complex change instances.

Instances. The instances of the defined schema are simple and complex change instances that are detected between two dataset versions. They are actually instances of the defined classes and are attributed with the defined properties.

Figure 3 presents an outline of the structure of the proposed RDF(S) schema. Two simple change classes and two complex change classes are presented indicatively, as well as a complex instance of Add_Academic_Professional presented in the running example.

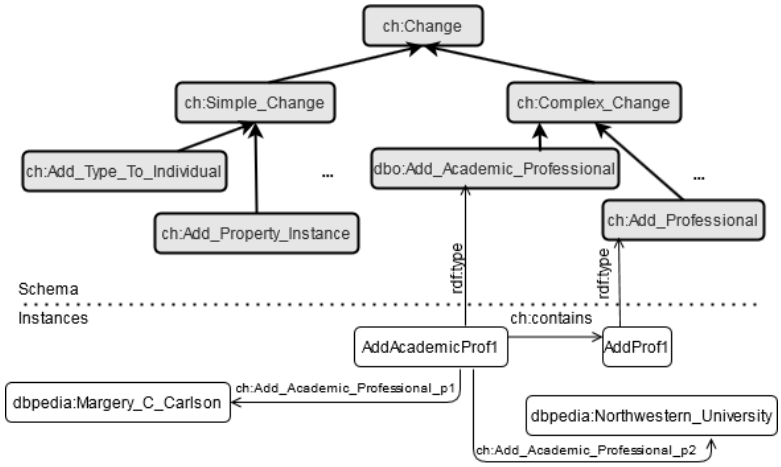


Figure 3 Outline of the proposed RDF(S) change representation.

3.5.3. SPARQL Query Generation

Algorithm 2 presents the process of generating a SPARQL query q, given a complex change c. This involves the generation of a *SELECT clause* (lines 2-6), a *FROM clause* (line 7), a *WHERE clause* (lines 8-17, 24-69), an *ORDER BY clause* (lines 18-21), and their concatenation into the query string q (line 22), which is finally returned (line 23). Note that prefix definitions should be defined, while omitted for simplicity.

SELECT Clause. Since the generated query is used for calculating complex change instances, it has to return: (1) the values to be assigned in the complex change descriptive parameters, (2) the change instances that are to be contained by the newly detected complex change instance. Thus, SELECT clause includes: (1) variables for the descriptive parameters (P) of the complex change as defined in the change heading statement (lines 3-4), (2) variables for identifying each change in the change list statement (lines 5-6). If c includes bindings, they are considered when specifying the variables for descriptive parameters via getBindParameter function call: each parameter p is substituted by the identifier used for its evaluation.

FROM Clause. The FROM clause includes two named graphs: S_i holding simple change instances and I holding complex change instances that are already computed (line 7).

WHERE Clause. Overall, the WHERE clause includes a graph pattern with: (1) triple patterns corresponding to the changes in the change list statement, (2) appropriate statements for the filters defined in the filter list statement. As for (1), the triple patterns that correspond for each change follow the RDF(S) representation defined in Section 3.5.2 and are generated via getTriplePattern function (lines 49-64). getTriplePattern requires as input a change d and a set of $filters$. Recall that a change parameter may be defined as optional (i.e., evaluating into empty value). The relevant triple pattern includes an optional graph pattern for this parameter, while unary filters on it are considered within the optional pattern (lines 51, 59-63). As for (2), each filter type is mapped to an appropriate SPARQL FILTER statement or subquery via getFilterPattern function (lines 65-69). getFilterPattern requires as input a filter f_c and the $changes$ over which the filter applies. Testing value, relational and functional constraints, as well as pre-/post-conditions on scalar parameters are similar to built-in SPARQL constraints.

However, triple patterns corresponding to changes and filters must be structured in an appropriate manner due to composite filter expressions and cardinality constraints. Recall that filters may be combined into logical expressions using logical AND, OR, NOT. In this case, the equivalent DNF (disjunctive normal form) of the expression is computed (line 9). Each conjunction is a combination of filters that should be satisfied by the changes in the change list. Therefore, the WHERE clause is formed as the union of graph patterns where each includes the appropriate triple patterns for changes plus one of the possible combination of filters (lines 10-17). Each such graph pattern is generated by a getPattern function call (lines 11, 14), where getPattern function (lines 24-37) together with getOptionalPattern (lines 38-48), for handling optional changes, orchestrate the process.

As for getPattern function, it requires as input a change c and a set of filters constituting a filter conjunction. First, the mandatory (cardinality 1 or "+") and optional (cardinality "?" or "*") changes are identified (lines 25-26). Second, the triple pattern for each mandatory change is generated (lines 27-28) via getTriplePattern function call, as well as the pattern for each unary or binary filter on mandatory changes (lines 29-34) via getFilterPattern function call. Notice that the getFilterPattern function call has as input the filter f_c and the changes that its parameters are on, generated by the getChanges function call. Next, the triple patterns for optional changes are considered (lines 35-36), starting from those that are directly connected via a filter f_c to mandatory changes (line 36), generated by getOptionalPattern function call (line 36). Finally, getPattern function returns the generated pattern (line 37).

As for getOptionalPattern function, it is a recursive function that generates a SPARQL optional statement, which may contain nested SPARQL optional statements, following the dependencies between optional changes, ultimately forming optional change paths (see Section 3.4.2.1). getOptionalPattern requires as input, an optional change d, a set of filters ($filterConjunction$), a set of changes of previous iteration ($parentChanges$, that d is dependent) and a set of optional changes ($optionalChanges$). First, the triple pattern for the optional change d is generated into a SPARQL optional statement (line 38). Second, for each filter f_c, that is unary with its parameter on d or binary with its parameters on d and $parentChanges$, a triple pattern is generated within the SPARQL optional statement via a getFilterPattern function call (lines 39-44). Next, it is examined if there are optional changes d' dependent on d and for each such change getOptionalPattern function is recursively called, resulting in the generation of a nested SPARQL optional statement (lines 45-46). Finally, the optional pattern is completed (line 47) and then returned (line 48).

ORDER BY Clause. Complex change detection is a two step process, where the second step is change instance generation based on the SPARQL query results. In order to facilitate this step it is necessary to have the query results in order, so that results that are to be grouped into one complex change instance are positioned nearby in the query result set. Thus, an ORDER BY clause (lines 18-21) is considered with: (1) *grouping variables* except from those representing set parameters, (2) *change variables* for each change in the change list statement. Based on the semantics presented in Section 3.4.2.2, *grouping variables* are the ones in *variable list*, which defines the groups, and are further discussed in Section 3.5.4. The variables representing set parameters are excluded, because set values "span" among multiple lines in the result set. In order to be computed, ordering based on all change variables is needed.

Algorithm 2: SPARQL Query Generation

Input: A complex change $c = (n, P, D, F)$ (where $F = F_C^{car} \cup F_C^{par} \cup F_B$, F_C^{car} is a set of cardinality constraints, F_C^{par} filter constraints, F_B bindings), a named graph of simple change instances S_i, a named graph of complex change instances I, a named graph of the version before V_{bef}, and a named graph with the version after V_{af}

Output: A SPARQL query q

$q \leftarrow$ "" ;
$selectClause \leftarrow$ "SELECT " ;
for each p **in** P **do**
  $selectClause \leftarrow selectClause +$ " ?" $+ getBindParameter(p, F_B)$; **end for**
for each d **in** D **do**
  $selectClause \leftarrow selectClause +$ " ?" $+ d$; **end for**
$fromClause \leftarrow$ " FROM <" $+ S_i +$ "> FROM <" $+ I +$ "> " ;
$whereClause \leftarrow$ " WHERE {" ;
$dnfFilterExpr \leftarrow Dnf(F_C^{par})$; // compute equivalent DNF expression
if ($dnfFilterExpr$ *is a* $filterConjunction$) **do** // generate a triple pattern
  $getPattern(c, dnfFilterExpr)$;
else // generate a union of triple patterns
  for each $filterConjunction_i$ **in** $dnfFilterExpr$ **do**
    $whereClause \leftarrow whereClause +$ "{" $+ getPattern(c, filterConjunction_i) +$ "}" ;
    if ($dnfFilterExpr.size() > i$) **do** $whereClause \leftarrow whereClause +$ " UNION " ;
end if end for
end if
$whereClause \leftarrow whereClause +$ "}" ;
$orderByClause \leftarrow$ " ORDER BY" ;
for each d **in** D **do** $orderByClause \leftarrow orderByClause +$ " ?" $+ d$; **end for**
for each p **in** P **do**
  if $((isInInferredBinding(p, F_B) \vee isInBindingWithoutAggregation(p, F_B)) \wedge isOnlyOnChangesWithCardinality + Or * (p, D, F_C^{car}) \wedge evaluatesIntoScalar(p))$ **do** $orderByClause \leftarrow orderByClause +$ " ?" $+ p$; **end if end for**
$q \leftarrow selectClause + fromClause + whereClause + orderByClause$;
return q ;

getPattern $(c, filterConjunction)$

$pattern \leftarrow$ "" ;
$mandatoryChanges \leftarrow getMandatoryChanges(D, F_C^{car})$;
$optionalChanges \leftarrow getOptionalChanges(D, F_C^{car})$;
for each d **in** $mandatoryChanges$ **do**
  $pattern \leftarrow pattern + getTriplePattern(d, filterConjunction)$; **end for**
for each $f_C \in filterConjunction$ **do**
  if $(isUnary(f_C) \wedge isOnChanges(f_C, mandatoryChanges))$ **do**
    $pattern \leftarrow pattern + getFilterPattern(f_C, getChanges(f_C, mandatoryChanges))$;
  else if $(isBinary(f_C) \wedge isOnChanges(f_C, mandatoryChanges))$ **do**
    $pattern \leftarrow pattern + getFilterPattern(f_C, getChanges(f_C, mandatoryChanges))$;
  end if end for
for each d **in** $optionalChanges$ **do** // generate optional pattern
  if $\exists f_C \in filterConjunction\ s.t.\ isOnChanges(f_c, d, mandatoryChanges)$ **do** $pattern \leftarrow pattern + getOptionalPattern(d, filterConjunction, mandatoryChanges, optionalChanges)$; **end if end for**
return $pattern$;

getOptionalPattern $(d, filterConjunction, parentChanges, optionalChanges)$

$optionalPattern \leftarrow$ " OPTIONAL{" $+ getTriplePattern(d, filterConjunction)$;
for each $f_C \in filterConjunction$ **do**
  if $(isUnary(f_C) \wedge isOnChanges(f_C, \{d\}))$ **do**
    $optionalPattern \leftarrow optionalPattern + getFilterPattern(f_C, \{d\})$;

```
42  else if (isBinary(f_C) ∧ isOnChanges(f_C, d, parentChanges)) do
43   optionalPattern ← optionalPattern + getFilterPattern(f_C, getChanges(f_C, {d} ∪
parentChanges)) ;
44  end if end for
45 for each d′ in optionalChanges do // generate nested optional pattern
46  if ∃f_C ∈ filterConjunction s.t. isOnChanges(f_c, d′, {d}) do optionalPattern ←
optionalPattern + getOptionalPatthern(d′, filterConjunction, {d}, optionalChanges) ; end
if end for
47 optionalPattern ← optionalPattern + "}" ;
48 return optionalPattern ;
getTriplePattern(d, filters) // where d = (name_d, P_d)
49 triplePattern ← " ?" + d + " rdf:type" + " " + predicate + ":" + name_d + ";" ; //
change type
50 nonEmptyParameters ← getNonEmptyParameters(P_d) ;
51 emptyParameters ← getEmptyParameters(P_d) ;
52 for each p_i ∈ nonEmptyParameters do // change parameters with non-
empty values
53  triplePattern ← triplePattern + " ch:" + name_d + "_p" + i + " ?" + p_i ;
54  if (nonEmptyParameters.size() > i) do
55   triplePattern ← triplePattern + ";" ;
56  else
57   triplePattern ← triplePattern + "." ;
58  end if end for
59 for each p_i ∈ emptyParameters do // optional change parameters, with
empty values
60  triplePattern ← triplePattern + " OPTIONAL{ ?" + d + " ch:" + name_d + "_p" + i +
" ?" + p_i ;
61  for each f_C ∈ filters s.t. isOnChangeParameter(f_c, d, p_i) do // unary filters
62   triplePattern ← triplePattern + getFilterPattern(f_C, {d}) ; end for
63  triplePattern ← triplePattern + "}" ; end for
64 return triplePattern ;
getFilterPattern(f_C, changes) // f_C is mapped to appropriate statement
65 testing value constraint on scalar parameter → SPARQL FILTER statement
66 relational constraint on scalar parameters → SPARQL FILTER statement
67 pre/post − condition on scalar parameters → FILTER EXISTS/NOT EXISTS on V_bef/V_af
68 function constraint → SPARQL built − in functions are considered
69 constraints on set parameters or quantification → SPARQL sub − queries and MINUS
```

As an example, consider the complex change Add_Academic_Professional defined in Section 3.4.3, example 5. Table 2 presents the SPARQL query for the detection of this complex change. In the SELECT clause notice the query variable corresponding to change's identifier (?c_1) and the query variables corresponding to the complex change's descriptive parameters (?id, ?E). In the FROM clause, the named graph S_i holds the simple change instances and the named graph I holds the complex change instances. In the WHERE clause, notice the triple pattern for the Add_Professional change defined in change list. For the post-condition an appropriate SPARQL filter expression is considered evaluating over the named graph holding V_{af}. Since it involves quantification, it is implemented via MINUS and a nested query. Finally, notice the ORDER BY clause which involves the contained change identifier.

Table 2 SPARQL query for the detection of complex change Add_Academic_Professional

```
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX dbo: <http://dbpedia.org/ontology/>
PREFIX ch:  <http://dblab.ece.ntua.gr/change#>
SELECT ?c_1 ?id ?E
FROM <Si> FROM <I>
WHERE{?c_1 rdf:type dbo:Add_Professional; ch:Add_Professional_p1 ?id;
           ch:Add_Professional_p2 ?E; ch:Add_Professional_p3 ?t;
           ch:Add_Professional_p4 ?sY; ch:Add_Professional_p5 ?eY.
   MINUS{ SELECT ?c_1
          WHERE{?c_1 rdf:type dbo:Add_Professional;
                     ch:Add_Professional_p2 ?e.
                FILTER NOT EXISTS {
                       GRAPH ?g { ?e rdf:type dbo:University. }
                       FILTER (?g=<Vaf>) } } }
} ORDER BY ?c_1
```

3.5.4. Change Instance Generation

In order to generate change instances, the SPARQL query result set is read line-by-line and the lines that share the same values in the *grouping variables* are used to form a new complex change instance. The new complex change instance contains the change instances bound to the change variables and it is described by the values bound to the variables of the descriptive parameters, that correspond to the result set lines to be grouped.

Grouping variables indicate the groups that have to be defined over the result set. Based on the semantics presented in Section 3.4.2.2, they actually form the variable list for group construct and comprise of: (1) the change variables of changes with cardinality 1 or "?" and (2) the descriptive variables that are used in assignments without aggregation and correspond (only) to changes with cardinality "+" or "*". The result set lines that share common values in the grouping variables are used to generate a new complex change instance.

For example, consider the complex change Add_Academic_Professional presented in Section 3.4.3, example 5 and the respective SPARQL query in Table 2. Add_Professional has cardinality one and there are not any complex change descriptive parameters coming only from changes with cardinality "+" or "*" used in bindings without aggregation. Thus, ?c_1 is the grouping variable. As another example, consider the complex change Add_Professionals_ withCommon_Employers presented in Section 3.4.3, example 6. Add_Professional has cardinality "+" and the complex change descriptive parameter *E* is repeated (only) on it

implying an inferred binding without aggregation. Given an appropriately generated SPARQL query, variable ?E (corresponding to parameter *E*) is the grouping variable.

Algorithm 3 presents the process of generating complex change instances. Grouping variables may involve scalar and set parameters. Thus, while iterating the result set the grouping variables of type set have to be calculated. For this reason, *ordering variables* are used: the variables over which the result set has been ordered (as in Section 4.5.3 ORDER BY clause).

Algorithm 3: Complex Change Instance Generation for grouping variables corresponding to scalar and set parameters

Input: A result set rs of a SPARQL query of complex change c, the set of grouping variables P of c

Output: A set of complex changes instances I_c of c

```
1  I_c ← { } ;
2  currOrdervals ← { } ;// ordering variable values
3  prevOrdervals ← { } ;// ordering variable values of previous iteration
4  currPvals ← { } ; // grouping variable values
5  prevPvals ← { } ; // grouping variable values of previous iteration
6  g_i ← createNewGroup() ; // group of change instances and descriptive
variables to be used in complex change instance generation
7  while rs.hasNext() do
8   r ← rs.next() ;
9   prevOrdervals ← currOrdervals ;
10  currOrdervals ← getOrdervals(r,P) ; // calculate current order values
11  if (currOrdervals = prevOrdervals) do
12   currPvals ← updatePvals(currPvals,r) ; // r forms the current grouping
values
13   g_i ← updateGroup(c_i,r) ; // r forms the current g_i
14  else
15   I_c ← updateI_c(g_i,currPvals,I_c) ; // g_i 's computation is completed, the
relevant c_i has to be updated / formed in I_c based on g_i
16   currPvals ← createNewPvals(r,P) ; // r forms the new current currPvals
17   g_i ← createNewGroup(r) ; // r forms the new current g_i
18  end if
19  if (!rs.hasNext()) do
20   I_c ← updateI_c(g_i,currPvals,I_c) ; // update I_c for the last iteration
21  end if
22 end while
23 return I_c ;
updateI_c(g_i,currPvals,I_c)
24 if (existInstanceWith_currPvals(I_c,currPvals)) do // there is a complex
change instance with the same grouping values
25  c_i ← getInstanceWith_currPvals(I_c,currPvals) ;
26  I_c ← I_c \ {c_i} ; // exclude c_i
27  c_i ← updateInstance(c_i,g_i) ; // update the instance with change
instances and descriptive variables in g_i of the current iteration
28  I_c ← I_c ∪ {c_i} ; // add updated c_i
29 else
30  c_i ← createNewInstance(g_i) ; // g_i forms a new complex change instance
31  I_c ← I_c ∪ {c_i} ; // add the newly created c_i
32 end if
33 return I_c ;
```

The result set is iterated and each time a new line is read, the values of the ordering variables are calculated: the current values ($currOrdervals$) and previous values ($prevOrdervals$, the values of previous iteration - line) (lines 7-10). If they are equal, the current grouping variable values have to be updated ($currPvals$), so that variables corresponding to set parameters can be computed (line 12). Accordingly, a variable holding the currently grouped data is updated (g_i) (line 13): it holds change instances and descriptive values that are considered in a complex change instance (c_i). Otherwise, the current grouping variables ($currPvals$) computation has finished. This is ensured by having the query result set in order. The appropriate complex change instance (c_i) in the result (I_c) has to be updated (line 15) via function $updateI_c$. The currently read result set line (r) is to be used to form new current grouping variable values ($currPvals$) and new current group data (g_i) (lines 16-17). If the result set has been read, the lastly computed grouping variable values and group data should be used to update the appropriate complex change instance of the result (I_c) (lines 19-21). Finally, the set of all computed complex change instances is returned (line 23).

Regarding the function $updateI_c$: it takes as input a set of grouping variable values ($currPvals$) and the respective grouped data (g_i), and returns the updated result (I_c). If there is already a complex change instance (c_i) with the same grouping variable values ($currPvals$) in the result (I_c), then the grouped data (g_i) have to be included within it (lines 24-28). Otherwise, a new complex change instance is created and added in the result (I_c) (lines 29-32). Finally, the set of all computed complex change instances is returned (line 33).

3.5.5. Complex Change Detection Correctness

Below we prove the correctness of the detection algorithm in Section 3.5 with respect to complex change language semantics. First, a subset of the proposed language is proven to have equivalent semantics to a subset of SPARQL. SPARQL semantics are defined in Perez, Arenas and Gutierrez (2009) [46] and Kaminski, Kostylev and Cuenca Grau (2017) [32]. Next, augmenting with the rest features, semantics are implemented by applying Algorithm 3 to the result mappings of a SPARQL graph pattern.

Step 1. Consider the subset of the proposed complex change language which involves only changes with cardinalities one and "?", scalar parameters and filter expressions on scalar parameters. Complex change semantics are defined given a set of change instances I and SPARQL semantics given an RDF graph D. Let D contain the RDF representation of I based on the vocabulary presented in Section 3.5.2.

(1) The abstract syntax of the proposed language is by definition equivalent to the one proposed for SPARQL in Perez, Arenas and Gutierrez (2009) [46], assuming that a graph pattern involves triples for changes, except that: (a) UNION operator is not considered, (b) the right operand of OPT shall be a graph pattern corresponding to a primitive change pattern, or a filter primitive change pattern, or an optional change pattern involving only primitive change patterns, filter primitive change patterns or optional change patterns with these types of operands, (c) the right operand of OPT may be a triple that involves an optional variable $xOPT$ (recall, if $xOPT \in dom(\mu_c)$ then $\mu_c(xOPT) = \emptyset$ or $\mu_c(xOPT) \neq \emptyset$). All complex change language's built-in filter expressions are SPARQL built-in filter expressions as well. For a complete SPARQL feature list see Harris S. and Seaborne A. (2013) [31].
(2) The semantics of the proposed language are by definition equal to SPARQL semantics as in Perez, Arenas and Gutierrez (2009) [46] for the syntax in (1), since they are made up of semantically equivalent operators applied on equivalent data in the same sequence.
Algorithm 3 (grouping variables are the change variables) materializes the change instances, performing a trivial grouping, where each SPARQL result mapping forms a trivial group and a new complex change instance. Overall, $[\![change\ pattern]\!]_I = Algorithm3([\![graph\ pattern]\!]_D)$.

Step 2. Augment step 1 with set parameters. Consider a change pattern with a set variable X and a set of mappings μ_c, Ω_c. Since SPARQL does not support this feature, the graph pattern corresponding to the change pattern involves a scalar variable x corresponding to X. Evaluating the graph pattern results in a set of mappings μ, Ω. It holds that $dom(\mu_c) - \{X\} = dom(\mu) - \{x\}$. Based on step 1, for each $\mu_c \in \Omega_c$ there is a $\mu \in \Omega$ such that $\mu_c(y) = \mu(y)$ where $y \in dom(\mu_c) - \{X\}$. Based on μ_c definition for a set parameter $\mu_c(X) = \cup_{i=1,\dots,n} \mu_i(x)$, considering all μ_i where $\mu_c(y) = \mu_i(y)\ \forall y \in dom(\mu_c) - \{X\}$ or simply $\forall y \in dom(\mu_c) - \{X\}$ and y is a change variable. Optional set variables are handled similarly. Therefore, the complex change semantics equal SPARQL semantics for step 1 plus Algorithm 3 for implementing set variable semantics: $[\![change\ pattern]\!]_I = Algorithm3([\![graph\ pattern]\!]_D)$.

Step 3. Augment step 2 with filter expressions on set parameters. These expressions are not SPARQL built-in expressions. Thus, each such expression R is mapped to an equivalent R' in SPARQL, based on built-in features (FILTER EXIST/NOT EXIST, MINUS and subqueries). The exact mapping for each one filter expression into SPARQL is not discussed in further detail. Also, R may combine primitive filter expressions with logical connectives. In this case, there is always an equivalent DNF expression $DNF(R) = R_1 \vee R_2 \vee \dots \vee R_n$. Since,

$[\![P\ FILTER\ R]\!]_I = \{\mu \in [\![P]\!]_I | \mu \vDash R\} = \{\mu \in [\![P]\!]_I | \mu \vDash R_1 \vee R_2 \vee ... \vee R_n\}$ and $[\![P\ FILTER\ R_1]\!]_I = \{\mu \in [\![P]\!]_I | \mu \vDash R_1\}$, ..., $[\![P\ FILTER\ R_n]\!]_I = \{\mu \in [\![P]\!]_I | \mu \vDash R_n\}$, it is implied that $[\![P\ FILTER\ R]\!]_I = [\![P\ FILTER\ R_1]\!]_I \cup ... \cup [\![P\ FILTER\ R_n]\!]_I$. Thus, $P\ FILTER\ R$ can be mapped in SPARQL to the union of all graph patterns where each comprises of P and Ri.

Overall, the complex change semantics are equal to the semantics of an equivalent SPARQL graph pattern plus Algorithm 3 for implementing the semantics of set variables (as in step 2). Again, $[\![change\ pattern]\!]_I = Algorithm3([\![equivalent\ graph\ pattern]\!]_D)$.

Step 4. Augment step 3 with cardinalities "+" and "*" and with union aggregation function. The change pattern is in extended form, including groups and aggregation. In Definition 12, a group $\Gamma = Group(V_r^g, P)$ is defined over a change pattern P and a list of variables V_r^g (grouping variables). In Definition 13, an aggregate is a construct of the form $A = Aggregate(v_r, union, \Gamma)$ where v_r is a variable over which $union$ aggregate function is performed for each group Γ. Based on previous steps, P is mapped to a SPARQL graph pattern P', such that $[\![P]\!]_I = Algorithm3([\![P']\!]_D)$ (3). Groups and aggregation computation is based on variables in V_r^g, which is by definition a superset of the variables used by Algorithm 3 in (3), since in previous steps the grouping variables are the change variables. Thus, $[\![A]\!]_I = Algorithm3([\![P']\!]_D)$ and grouping variables are those in V_r^g. Union aggregation function is implemented by Algorithm 3, also implementing set variable semantics for computing set grouping variables.

3.6. Evaluation

The proposed approach has been evaluated qualitatively and experimentally. In qualitative evaluation, our approach is compared to the related work discussed in Chapter 2, Section 2.1. In experimental evaluation, complex change language expressiveness and detection performance are examined. It is evaluated whether the proposed structures are adequate in expressing useful changes and how complex changes facilitate user in analyzing evolution. Also, the response time of the detection process is examined in terms of increasing dataset size.

3.6.1. Qualitative Evaluation

Our approach focuses on human readable changes. Similar to Klein (2004) [33], Stojanovic (2004) [57], Papavasileiou et al. (2013) [45] and Roussakis et al. (2015) [53] we assume primitive changes, as simple changes, and groupings of them, as complex changes. Instead of providing a predefined list of complex changes, we support user-defined complex changes in order to capture richer semantics and multiple interpretations of evolution, as Plessers, De Troyer and Casteleyn (2007) [47] and Roussakis et al. (2015) [53]. In our approach, a dedicated complex change definition language is formally defined, so that complex changes are defined via patterns, and an appropriate detection algorithm is proposed. Instead, Plessers, De Troyer and Casteleyn (2007) [47] relies on temporal queries and Roussakis et al. (2015) [53] on SPARQL in order to define and detect changes. On top of this, we support relations and dependencies among complex changes, so that complex changes may share common parts.

The closest relevant works to the proposed approach are Papavasileiou et al. (2013) [45] and Roussakis et al. (2015) [53]. The proposed notion of complex changes resembles to the "composite changes" presented in Papavasileiou et al. (2013) [45] in their ultimate goal in grouping changes into logical units. But, complex changes are user defined and may be related to each other, providing richer semantics and flexibility. In Roussakis et al. (2015) [53] the notion of complex changes as user defined is also stated. There, the proposed changes may not share common parts but instead are given a prioritization. However, prioritization possibly leads to the loss of part of the evolution interpretation, when two changes are identified simultaneously over a data element. On the contrary, by allowing interdependencies among complex changes all possible interpretations are maintained. Towards this direction, a complex change may be defined on top of another. In this case, the process of defining new complex changes is facilitated by reusing already defined patterns. Also, in Roussakis et al. (2015) [53] the complex changes are defined via SPARQL queries. However, for supporting the reusability of changes, each change pattern should be given a specific name and descriptive properties. In addition, it may be needed to define explicitly how a complex change groups possible multiple appearances (instances) of changes in its definition, either by following the underlying data structure or the current understanding on modeling evolution. Although SPARQL is powerful in defining patterns over RDF data, it does not provide such capabilities. Thus, a dedicated language for defining complex changes and a relevant detection algorithm are needed.

Table 3 summarizes the qualitative comparison of our approach with the most significant works on high-level changes. It is examined whether predefined or user-defined changes are supported, if a detection algorithm is presented and the data model each approach focuses on. For works that support user-defined changes, several features are further examined.

Table 3 Qualitative comparison of this approach with related work

	Predefined Changes	User-Defined Changes	User-Defined Changes Features			Detection Algorithm	Data Model
			Dedicated Language	Relations among Changes	Cardinality/ Grouping		
Klein (2004) [33]	x	-	-			x	OWL/OKBC
Stojanovic (2004) [57]	x	-	-			- (change application)	KAON
Plessers et al (2007) [47]	-	x	x (temporal logic based)	-	-	x	OWL DL
Papavasileiou et al (2013) [45]	x	-	-			x	RDF(S)
Rousakis et al. (2015) [53]	x	x	- (SPARQL queries)	-	-	x	RDF(S)
This approach	x	x	x	x	x	x	RDF(S)

3.6.2. Experimental Evaluation

3.6.2.1. Implementation, datasets and settings

The complex change definition language and the detection process are implemented in a Java application. In order to implement the language parser JavaCC[2], a parser generator for Java, is employed. In order to store the RDF(S) representations of changes and change instances and run SPARQL queries for complex change detection Openlink Virtuoso[3] is employed. The implementation is done in Java version 8 and Openlink Virtuoso version 7.

In order to test the proposed approach, dataset versions and the respective simple changes, capturing the modifications between them, are required. The evaluation is performed over both artificial and real data. Artificial data are generated by the tool EvoGen[4], while DBpedia[5] dataset versions are considered for real data. As for the system settings, a 6-core CPU and 16 GB RAM machine running Ubuntu (version 16.04) has been used in order to host both Virtuoso server and the application.

[2] https://javacc.org/
[3] https://virtuoso.openlinksw.com/
[4] https://github.com/mmeimaris/EvoGen
[5] https://wiki.dbpedia.org/

EvoGen is a tool for generating synthetic evolving RDF datasets, abstracting several characteristics of the process (Meimaris (2016) [39]; Meimaris and Papastefanatos (2016) [40]). It extends the Lehigh University Benchmark (LUBM) generator (Guo et al. (2005) [30]), a Java based synthetic data generator, which features an ontology for the university domain called Univ-Bench. An OWL[6] version of the Univ-Bench ontology[7] is available in OWL Lite[8].

In this evaluation, only data changes are employed. Also, the current version of EvoGen generates changes that include only additions. It creates a log with the changes between consecutive versions, following the simple changes paradigm that the proposed complex changes rely on. As a result, complex changes for this experiment involve only additions.

Table 4 presents the sizes of the RDF datasets generated with EvoGen. The sizes of the simple changes log between two consecutive versions are presented, in terms of number of triples and number of simple change instances. Also, the sizes of the version before and version after, in terms of number of triples, are presented.

Regarding DBpedia data, three previous DBpedia releases have been considered: versions 2016-10, 2016-04 and 2015-10. Specifically, parts of the English DBpedia datasets are considered, namely the instance types and mapping based objects. First, the detection of simple changes took place among dataset versions, and then a number of complex changes were defined on top involving both additions and deletions. The complex changes defined focus on data changes. Table 5 presents the sizes of the RDF datasets of DBpedia data.

Table 4 EvoGen generated datasets

Dataset	Simple Change Log (# of triples)	Simple Change Log (# of change instances)	Version Before (# of triples)	Version After (# of triples)
D0	212.178	53.072	99.761	150.836
D1	473.955	118.550	220.840	334.829
D2	1.489.892	372.667	690.550	1.048.536
D3	8.246.486	2.062.693	3.778.293	5.759.845
D4	27.882.797	6.974.311	12.753.945	19.454.837
D5	41.465.290	10.371.705	19.013.429	28.978.776
D6	83.041.295	20.771.095	37.990.459	57.948.069

[6] https://www.w3.org/TR/2012/REC-owl2-overview-20121211/
[7] http://swat.cse.lehigh.edu/onto/univ-bench.owl
[8] https://www.w3.org/TR/owl-features/

Table 5 DBpedia datasets

Dataset	Simple Change Log (# of triples)	Simple Change Log (# of change instances)	Version Before Name	Version Before (# of triples)	Version After Name	Version After (# of triples)
Δ0	9.198.606	2.449.083	2015-10	22.841.862	2016-04	23.401.677
Δ1	29.911.620	7.755.452	2016-04	23.401.677	2016-10	23.896.605

3.6.2.2. Language expressiveness

In order to evaluate the expressiveness of the proposed language several complex changes have been defined for the data generated with EvoGen and for the DBpedia data. The Univ-Bench ontology and the DBpedia schema have been studied in order to identify classes, descriptive properties and how data are connected. In order to define complex changes that are as realistic as possible the process below was followed, identifying six cases of possible complex changes, based on common sense and domain characteristics.

1. Class instance additions/deletions: For each class instance addition/deletion a descriptive change should be reported based on the dataset domain. Therefore, a complex change with a descriptive name of each class is defined, being actually a rename of the respective simple change. For example, Add_Person instead of Add_Type_To_Individual.

2. Property instance additions/deletions: Similarly, for each property instance addition/ deletion, a complex change with a descriptive name of the property is defined. This is a rename of the respective simple change, Add_Property_Instance/Delete_Property_Instance.

Notice that complex changes of cases 1 and 2 may form a first level of complex changes over the simple changes of a dataset, where each simple change is mapped to a complex change with a more descriptive name based on the specific domain.

3. Groupings around added/deleted class instance URIs: Typically, a class instance addition/ deletion is accompanied with its property instances additions/deletions. Therefore, a complex change may be defined for grouping these changes altogether. Properties' cardinality should be considered accordingly: multiple instances of properties should be grouped altogether, while optional properties are allowed. Thus, added/deleted properties around a class instance URI are grouped into a complex change together with the added/deleted class instance. These complex changes are defined on top of the complex changes described in cases 1 and 2. For example, Add_Person_with_Details may group Add_Person along with all its added descriptive properties (name, birth date, death date, etc).

4. Batch additions/deletions: Complex changes of case 3 may appear in batches. In such case, they may share common values in some of their properties. Therefore, they can be further grouped based on these values. For example, Add_Professionals_withCommon_Employers groups all Add_Professional for those having the same employers.

5. Specializations: Data and domain specific changes may be important in certain scenarios. Such changes may be captured by further combining the complex changes described in previous cases via relational filters, testing value filters, pre-/post-conditions, and optional cardinalities. For example, Add_Academic_Professional specializes Add_Professional where the added professional works only in academia, which is specified by a post-condition.

6. Updates: A property value update can be modeled as an addition plus a deletion of the specific property over a specific class instance URI. These complex changes are defined on top of the complex changes described in case 2.

Regarding the data generated with EvoGen, 65 complex changes have been defined following the above process. The changes involve only additions, due to the characteristics of the current version of EvoGen as already discussed. Table 6 summarizes the characteristics of the defined complex changes in terms of the features of the proposed language. The complex changes have been grouped into twelve categories, where each one has specific: change list size, cardinalities on changes in the change list, grouping variables' type, as well as filter types employed. Nested complex changes are defined, and the level of each change in the complex change hierarchy is stated: complex changes in level I are defined on top of simple changes only, in level II on top of complex changes in level I, and in level III on top of changes in level II. Table 6 shows that all proposed features have been used. The number of complex change definitions per category is presented, as well as per language characteristic.

Category C1 involves class instance and property instance addition renames (cases 1 and 2). Testing value constraints are used for identifying class and property types. Categories C2-C7 involve groupings around added class instance URIs (case 3). Relational filters are used and appropriate cardinalities are defined based on the data model. Categories C8-C9 involve groupings based on common property values (case 4), on scalar/set parameters respectively. Categories C10-C12 involve specializations (case 5). Pre-/post-conditions and relational filters are combined with quantification, since they involve set parameters. In C12 optional change paths are defined. Complex changes form three levels over simple changes.

Table 6 Categories and characteristics of the defined complex changes on EvoGen data

Categories	# of complex change definitions	Change List Size (# of changes)			Cardinality (type)				Grouping (variables type)		Filter (type)				level in complex change hierarchy
		<=3	4-6	=>7	1	?	+	*	scalar	set	testing value	relational	pre-/post-condition	quantification	
C1	30	x			x						x				I
C2	3	x			x							x			II
C3	1		x		x							x			II
C4	4			x	x							x			II
C5	5	x			x		x		x			x			III
C6	1		x		x	x						x			II
C7	2			x	x	x						x			II
C8	7	x					x		x						III
C9	1	x					x			x					III
C10	4	x			x		x		x			x		x	III
C11	3	x			x								x	x	III
C12	4			x	x	x		x	x				x		II
# total changes	65	53	2	10	57	7	17	4	20	1	30	20	7	7	65

Table 7 presents the number of complex change instances detected in each EvoGen generated dataset per complex change category. Change instances of all categories appear in all datasets, proving the effectiveness of the proposed methodology. Also, given an increasing dataset size the number of detected complex change instances increases too. Notice that the number of change instances in C1 for each dataset is very close to the number of simple change instances presented in Table 4. This is because C1 involves changes defined as in case 1 and 2, forming a first level of complex changes serving as renames of simple changes. The total number of change instances in C2-C7 for each dataset is significantly smaller than the number of change instances in C1. This is because C2-C7 involve changes defined on top of C1 grouping property changes around class instance URIs (as in case 3). These changes form a second level of complex changes which compresses the changes of first level. The number of change instances in C8-C11 for each dataset is even smaller, since those changes further group or specialize changes in C2-C7. C12 offers an alternative way of grouping some of the changes in C1. Table 10 presents the number of complex change instances per level in the change hierarchy, quantifying the size of each level. It is worth noting that the number of complex change instances in level I covers the 99.9% of the simple change instances, while in level II covers the 93% of the complex change instances in level I, and in level III up to 45% of the complex change instances in level II. The smaller number of complex change instances of

Table 7 Number of complex change instances per category detected in EvoGen generated datasets

	C1	C2	C3	C4	C5	C6	C7	C8	C9	C10	C11	C12
D0	52.999	2.054	960	327	1.509	3.994	2.267	2.222	1.087	313	248	10.158
D1	118.387	4.611	2.160	745	3.499	8.743	5.141	4.020	2.513	726	571	22.260
D2	372.150	14.733	6.880	2.364	11.037	27.004	16.316	9.853	7.801	2.318	1.803	69.294
D3	2.059.836	81.767	38.080	13.091	61.296	148.765	90.515	47.224	43.050	12.912	9.950	382.150
D4	6.964.680	275.488	128.400	43.912	205.993	506.734	304.876	153.159	143.868	43.371	33.432	1.299.058
D5	10.357.352	410.568	191.360	65.309	305.474	755.204	451.961	226.342	213.460	64.485	49.725	1.934.000
D6	20.742.372	821.895	382.960	130.665	611.034	1.513.220	905.170	450.645	426.893	128.926	78.744	3.876.042

level II contained in a complex change instance of level III is due to the fact that level III changes also involves specializations of level II changes. Overall, the resulting complex change instances reduce the effort of analyzing the data evolution as the user can easily zoom-in/-out on evolution detail by navigating on different levels of the complex change hierarchy.

Regarding the DBpedia data, 177 complex changes have been defined following the aforementioned process. Complex changes of add type, symmetric changes of delete type and update changes were defined. Table 8 summarizes the characteristics of the defined complex changes based on the proposed language features. The complex changes have been grouped into eleven categories, based on similar characteristics to the ones used for EvoGen data.

Table 8 Categories and characteristics of the defined complex changes on DBpedia data

Categories	# of complex change definitions	Change List Size (# of changes)		Cardinality (type)				Grouping (variables type)		Filter (type)			level in complex change hierarchy
		<=3	4-6	1	?	+	*	scalar	set	testing value	relational	quantification	
Ci	94	x		x						x			I
Cii	4	x		x							x	x	III
Ciii	6	x		x		x		x			x		II
Civ	12	x				x			x				III
Cv	2		x	x	x	x		x			x		II
Cvi	8		x	x		x	x	x			x		II
Cvii	4		x	x			x	x			x		II
Cviii	29	x		x							x		II
Cix	4	x		x		x	x	x			x		II
Cx	12	x		x						x		x	III
Cxi	2	x		x		x		x			x	x	III
# total changes	177	163	14	165	2	34	16	26	12	106	59	18	177

Category Ci involves class instance and property instance addition and deletion renames (cases 1 and 2). Categories Ciii, Cv, Cvi and Cix involve groupings around added or deleted

class instance URIs (case 3). Category Civ involves groupings based on common property values (case 4). Categories Cii, Cvii, Cx and Cxi involve specializations (case 5). In Cvii optional change paths are defined. Category Cviii involves update changes (case 6). Complex changes form three levels over simple changes.

Table 9 presents the number of complex change instances detected in each DBpedia dataset per complex change category. Again, change instances of all categories appear in datasets and as the dataset size increases, the number of detected complex change instances increases too. Recall that changes in Ci form a first level of complex changes serving as renames of simple changes. The number of change instances in Ci for each dataset reaches the 60% of the number of simple change instances presented in Table 5. Although the DBpedia schema is very large and diverse, the changes defined in Ci span across the most frequently changed parts of the datasets. The total number of change instances in Ciii, Cv, Cvi and Cix is smaller related to Ci, because they are defined on top of Ci forming a second level of changes and grouping property changes around class instance URIs (as in case 3), for the most frequently added/deleted entities. Also, several update operations (Cviii) appear in the examined datasets. The total number of the change instances in Civ, Cii, Cvii, Cx and Cxi which form a third level in the change hierarchy is even smaller, since they further group changes of the second level or specialize their meaning narrowing down the reported changes. Table 10 presents the number of complex change instances per change hierarchy level. It is worth noting that the number of complex change instances in level I covers the 60% of the simple change instances, while in level II covers up to the 68% of the complex change instances in level I, and in level III covers up to the 10% of the complex change instances in level II. The relatively low number of complex change instances of level II contained in a complex change instance of level III is due to the fact that level III changes are mostly specializations of level II changes. Similar to EvoGen results, the complex change hierarchy facilitates the user in analyzing the data evolution.

Table 9 Number of complex change instances per category detected in DBpedia datasets

	Ci	Cii	Ciii	Civ	Cv	Cvi	Cvii	Cviii	Cix	Cx	Cxi
Δ0	1.464.340	38	50.181	571	179	9.757	3.178	62.050	57.227	2.364	5.076
Δ1	4.708.482	370	90.708	915	327	19.584	4.956	1.313.776	104.687	4.431	10.218

Table 10 Number of complex change instances per level in hierarchy per EvoGen and DBpedia dataset

	EvoGen							DBpedia	
	D0	D1	D2	D3	D4	D5	D6	Δ0	Δ1
Level I	52.999	118.387	372.150	2.059.836	6.964.680	10.357.352	20.742.372	1.464.340	4.708.482
Level II	19.760	43.660	136.591	754.368	2.558.468	3.808.402	7.629.952	182.572	1.534.038
Level III	5.379	11.329	32.812	174.432	579.823	859.486	1.696.242	8.049	15.934

Annex B demonstrates some of the complex change definitions for the EvoGen generated data and the DBpedia data, as defined in terms of the experimental evaluation.

3.6.2.3. Detection performance

In order to evaluate the detection process performance, the detection time has been measured for the datasets generated with EvoGen with an input of 65 complex changes and for the DBpedia datasets with an input of 177 complex changes. As already stated, in order to store and query the datasets for complex change detection Openlink Virtuoso is used. Particularly, for each dataset, each simple change log, version before and version after are stored in a separate named graph. A dedicated named graph is employed for the complex change instances, which is gradually enriched with the complex change instances that are detected while the detection process progresses. The detection of each complex change relies on the already generated simple and complex change instances.

The detection time can split in four parts based on the detection algorithm presented: (1) *parse time* for computing the order of detection of the complex changes and parsing each definition, (2) *query execution time* for running each generated SPARQL query against the simple and complex change instances graphs, (3) *instance generation time* for parsing the query result set for computing instances and serialize them in a file, (4) *load instance time* for loading the generated instances stored in file into the complex change instances graph. The parse time is minor and thus omitted, since this process is performed in memory. Overall, the total detection time is presented, as well as the rest three parts as percentages of it. The number of added triples and respective complex change instances are presented as well.

It is worth noting that due to nested complex change definitions, all complex changes are not detected over the same initial dataset, since as the detection process progresses the complex change named graph increases in size. The detection process has been also evaluated in a slightly different setting: Each specific complex change has a dedicated named graph, so that only its respective instances are stored in it. In such a case each query generated for the detection process can rely only on the particular named graphs holding the minimum set of change instances required for the detection. In this setting, the results in detection time were similar to the ones presented and in some cases worse. This is mainly due to the larger load times observed. The query times were in some cases improved and the instance generation times were the same (as expected) since the algorithm does not change at this point.

Table 11 presents the total detection time for each EvoGen generated dataset D0-D6. This is the time needed to run the detection process for all the 65 complex changes. It also presents the total number of detected complex change instances and the total number of respective triples. It can be observed that as the dataset size increases the detection time increases too. In smaller datasets (D0-D2) the query execution time is dominant in the detection time, while in larger datasets (D3-D6) all three parts of the detection time contribute almost evenly.

Table 12 presents the total detection time per complex change category for each EvoGen generated dataset D0-D6, based on the categories presented in Section 3.6.2.2. It also presents the total number of added triples and detected complex change instances in each category. The differences in the characteristics among categories possibly affect only the query execution time. For example, in categories C4, C7 and C12 (involving complex changes with a bigger number of changes in their change list) and for small datasets tend to appear the higher query execution times. The instance generation time and load time are expected to be affected by the query result set size, i.e. the number of complex change instances appeared. Therefore, as the dataset size and the number of detected complex change instances increase, the instance generation time and load time increase accordingly. Also, notice that category C1 plays a rather dominant role in the total detection time, since despite of the simplicity of the change definitions, a large number of instances (up to 20,7M) are detected.

Table 13 presents the total detection time for each DBpedia dataset Δ0-Δ1. This is the time needed to run the detection process for all the 177 complex changes. It also presents the total number of detected complex change instances and the total number of respective triples. Again, the detection time is bigger for bigger datasets. Also, the load time is rather dominant in the detection time. This is mainly due to the significantly large number of change instances in category Ci, as shown in Table 14 below. Taking into consideration the rest categories only, query execution time is also significant, while instance generation time is rather low in categories with rather sparse change instances (for example Cii, Cv).

Table 14 presents the total detection time per complex change category for each DBpedia dataset Δ0-Δ1 based on the categories presented in Section 3.6.2.2. It also presents the total number of added triples and detected complex change instances in each category. The higher detection time appears in category Ci, since it involves a significantly larger number of instances compared to all other changes. Overall, the instance generation time and load time increases as the number of detected complex change instances increases too.

Table 11 Total detection time (seconds), number of added triples and number of detected complex changes instances for each EvoGen generated dataset

Dataset	Query Exec Time	Instance Gen Time	Load Time	Detection Time	Added Triples	Change Instances
D0	92,2%	2,4%	5,4%	192,3	472.293	78.138
D1	85,1%	4,7%	10,2%	215,9	1.050.692	173.376
D2	75,4%	8,5%	16,0%	351,1	3.292.481	541.553
D3	45,8%	21,1%	33,1%	737,3	18.209.825	2.988.636
D4	36,9%	27,8%	35,3%	2.144,8	61.616.525	10.102.971
D5	30,5%	30,4%	39,1%	2.717,3	91.635.866	15.025.240
D6	29,9%	30,1%	39,9%	5.583,1	183.408.890	30.068.566

Table 12 Total detection time (seconds), number of added triples and number of detected complex changes instances per complex change category for each EvoGen generated dataset

	Query Exec Time	Instance Gen Time	Load Time	Detection Time	Added Triples	Change Instances		Query Exec Time	Instance Gen Time	Load Time	Detection Time	Added Triples	Change Instances
	Category C1							Category C7					
D0	55,0%	15,5%	29,5%	16,7	202.394	52.999	D0	86,2%	2,8%	10,9%	4,5	35.115	2.267
D1	44,9%	17,3%	37,8%	27,0	452.148	118.387	D1	68,4%	6,0%	25,6%	6,9	79.439	5.141
D2	32,2%	22,3%	45,5%	54,9	1.421.303	372.150	D2	59,6%	12,4%	27,9%	12,9	252.286	16.316
D3	30,1%	25,5%	44,4%	232,1	7.867.126	2.059.836	D3	34,5%	22,1%	43,4%	37,2	1.400.235	90.515
D4	21,9%	30,6%	47,5%	795,9	26.599.310	6.964.680	D4	40,8%	24,7%	34,5%	123,4	4.716.410	304.876
D5	16,1%	34,6%	49,3%	933,8	39.555.006	10.357.352	D5	35,0%	24,9%	40,1%	176,3	6.991.135	451.961
D6	24,0%	30,4%	45,7%	2.091,9	79.215.578	20.742.372	D6	45,2%	21,7%	33,2%	401,6	14.002.744	905.170
	Category C2							Category C8					
D0	24,7%	13,8%	61,5%	0,4	10.270	2.054	D0	25,6%	18,5%	55,9%	0,7	14.787	2.222
D1	30,7%	14,8%	54,5%	1,0	23.055	4.611	D1	38,1%	20,8%	41,1%	1,9	31.514	4.020
D2	21,0%	15,2%	63,8%	3,3	73.665	14.733	D2	27,3%	25,5%	47,2%	5,3	93.620	9.853
D3	23,9%	21,4%	54,7%	11,7	408.835	81.767	D3	22,9%	34,2%	42,9%	23,6	506.709	47.224
D4	31,3%	19,2%	49,5%	32,0	1.377.440	275.488	D4	23,8%	43,5%	32,7%	60,7	1.692.807	153.159
D5	26,8%	22,0%	51,2%	51,7	2.052.840	410.568	D5	18,6%	44,8%	36,6%	100,4	2.509.925	226.342
D6	26,0%	24,0%	50,0%	99,2	4.110.459	821.895	D6	15,2%	52,5%	32,3%	214,1	5.018.940	450.645
	Category C3							Category C9					
D0	85,1%	2,6%	12,3%	1,3	10.560	960	D0	8,8%	71,2%	20,1%	0,6	5.776	1.087
D1	71,0%	5,0%	24,0%	1,8	23.760	2.160	D1	7,9%	71,3%	20,8%	1,5	13.476	2.513
D2	46,3%	12,9%	40,8%	3,7	75.680	6.880	D2	6,6%	75,1%	18,3%	4,4	42.146	7.801
D3	44,9%	19,0%	36,1%	14,9	418.880	38.080	D3	7,7%	77,7%	14,6%	23,4	234.173	43.050
D4	25,5%	28,7%	45,8%	37,0	1.412.400	128.400	D4	5,8%	83,9%	10,2%	73,0	785.423	143.868
D5	20,9%	30,2%	48,9%	46,8	2.104.960	191.360	D5	7,0%	75,2%	17,8%	120,7	1.164.599	213.460
D6	22,2%	33,9%	44,0%	91,1	4.212.560	382.960	D6	4,4%	56,8%	38,8%	320,1	2.330.050	426.893
	Category C4							Category C10					
D0	99,9%	0,1%	0,1%	154,1	6.425	327	D0	43,8%	13,7%	42,5%	0,3	4.213	313
D1	99,7%	0,1%	0,2%	154,4	14.607	745	D1	64,3%	10,8%	24,9%	1,2	11.282	726
D2	99,3%	0,1%	0,5%	219,4	46.404	2.364	D2	39,2%	22,9%	37,9%	2,6	36.918	2.318
D3	95,9%	0,7%	3,4%	195,4	257.185	13.091	D3	23,3%	27,2%	49,6%	12,0	213.938	12.912
D4	92,3%	3,4%	4,4%	310,5	862.366	43.912	D4	53,8%	19,9%	26,4%	41,5	725.331	43.371
D5	89,9%	3,9%	6,2%	281,1	1.282.555	65.309	D5	55,9%	20,7%	23,5%	63,7	1.077.589	64.485
D6	75,0%	8,2%	16,8%	223,7	2.565.557	130.665	D6	35,8%	36,8%	27,4%	96,0	2.158.640	128.926
	Category C5							Category C11					
D0	25,8%	27,2%	47,0%	0,6	11.824	1.509	D0	59,0%	10,7%	30,3%	0,2	1.607	248
D1	40,1%	22,1%	37,8%	1,7	27.439	3.499	D1	63,7%	12,3%	23,9%	0,4	3.721	571
D2	30,1%	24,1%	45,8%	4,7	86.321	11.037	D2	47,4%	15,1%	37,5%	1,0	11.708	1.803
D3	23,5%	29,5%	47,0%	18,8	480.158	61.296	D3	27,8%	21,1%	51,1%	4,1	64.667	9.950
D4	24,8%	43,5%	31,7%	54,6	1.612.359	205.993	D4	51,2%	26,5%	22,2%	27,7	217.455	33.432
D5	22,4%	39,0%	38,7%	78,1	2.392.209	305.474	D5	47,6%	22,0%	30,4%	28,8	323.282	49.725
D6	21,6%	43,3%	35,1%	143,9	4.785.690	611.034	D6	42,1%	22,3%	35,5%	44,4	511.696	78.744
	Category C6							Category C12					
D0	55,2%	7,8%	37,0%	2,9	45.574	3.994	D0	69,1%	6,5%	24,4%	10,1	123.748	10.158
D1	39,5%	15,5%	45,0%	4,3	99.589	8.743	D1	51,9%	13,6%	34,6%	14,0	270.662	22.260
D2	28,3%	23,8%	47,9%	8,6	307.778	27.004	D2	39,9%	20,6%	39,5%	30,2	844.652	69.294
D3	21,3%	27,3%	51,4%	40,6	1.696.073	148.765	D3	27,4%	27,8%	44,8%	123,7	4.661.846	382.150
D4	30,8%	28,5%	40,8%	144,1	5.776.808	506.734	D4	33,5%	28,4%	38,1%	444,5	15.838.416	1.299.058
D5	26,4%	28,1%	45,5%	204,1	8.609.346	755.204	D5	30,5%	28,6%	40,9%	631,8	23.572.420	1.934.000
D6	31,6%	27,0%	41,3%	427,4	17.250.720	1.513.220	D6	35,5%	25,9%	38,6%	1.429,8	47.246.256	3.876.042

Table 13 Total detection time (seconds), number of added triples and number of detected complex changes instances for each DBpedia dataset

Dataset	Query Exec Time	Instance Gen Time	Load Time	Detection Time	Added Triples	Complex Change Instances
Δ0	26,4%	24,0%	49,7%	578,9	6.636.996	1.654.961
Δ1	16,7%	27,0%	56,3%	1390,3	27.374.997	6.258.454

Table 14 Total detection time (seconds), number of added triples and number of detected complex changes instances per complex change category for each DBpedia dataset

	Query Exec Time	Instance Gen Time	Load Time	Detection Time	Added Triples	Complex Change Instances
Category Ci						
Δ0	16,1%	25,9%	58,0%	402,3	5.408.640	1.464.340
Δ1	8,8%	28,2%	63,0%	891,9	17.946.202	4.708.482
Category Cii						
Δ0	74,6%	0%	25,4%	0,3	190	38
Δ1	52,7%	5,1%	42,2%	0,3	1.850	370
Category Ciii						
Δ0	23,8%	26,7%	49,4%	25,5	251.077	50.181
Δ1	15,9%	32,2%	51,9%	37,8	454.502	90.708
Category Civ						
Δ0	40,1%	12,5%	47,4%	4,0	17.257	571
Δ1	9,7%	11,4%	78,9%	9,3	33.270	915
Category Cv						
Δ0	84,2%	1,8%	13,9%	0,5	1.279	179
Δ1	80,5%	1,9%	17,6%	0,7	2.176	327
Category Cvi						
Δ0	46,4%	37,1%	16,5%	46,4	169.993	9.757
Δ1	29,9%	45,8%	24,3%	77,5	323.319	19.584

	Query Exec Time	Instance Gen Time	Load Time	Detection Time	Added Triples	Complex Change Instances
Category Cvii						
Δ0	72,7%	4,1%	23,2%	1,2	9.543	3.178
Δ1	36,5%	12,3%	51,2%	0,9	14.878	4.956
Category Cviii						
Δ0	64,7%	10,2%	25,1%	59,2	372.300	62.050
Δ1	32,2%	21,6%	46,2%	316,5	7.882.656	1.313.776
Category Cix						
Δ0	49,7%	8,2%	42,1%	32,3	334.421	57.227
Δ1	39,7%	9,4%	50,8%	47,2	581.603	104.687
Category Cx						
Δ0	60,8%	2,1%	37,1%	2,6	7.092	2.364
Δ1	31,2%	5,4%	63,4%	2,0	13.293	4.431
Category Cxi						
Δ0	27,8%	22,1%	50,1%	4,5	65.204	5.076
Δ1	17,3%	30,0%	52,7%	6,3	121.248	10.218

3.6.2.4. Results summary

Overall, the detection times presented are considered acceptable since change detection is rather an off-line process executed upon version creation. It can be stated that as the dataset size increases, the number of detected change instances as well as the detection time increase too. Actually, the detection time is highly dependent on the number of instances that appear in the dataset. This can be verified by comparing in terms of total detection time the results of the experiment on EvoGen generated datasets with the results of the experiment on DBpedia datasets which are similar in size (D3-D4, Δ0-Δ1). The number of complex changes to be detected seems to have minor effect in the detection performance, since while in EvoGen 65 changes were used and in DBpedia 177 changes, the detection time is proportional to the number of detected change instances. Also, the complex change definitions' complexity has minor impact. In conclusion, the performance of complex change detection process is highly dependent on the dataset size over which detection is performed and the number of instances that appear in dataset, while it may be affected by the complex change definitions complexity.
In terms of detection process performance it is not possible to compare directly this work with the closest relevant works, Papavasileiou et al. (2013) [45] and Roussakis et al. (2015) [53], since the experimentation settings (datasets size, number of change instances, change definitions complexity) and the testing environments are diverse.

Chapter 4

Querying Data Versions and Change Structures on XML Data

4.1. Introduction

Apart from identifying human readable and semantically rich changes among dataset versions, querying data evolution may also provide insights on how data changed. In our view, querying evolution should be based on data as much as on changes. If changes are modeled as first-class-citizens, they can be exploited in terms of querying as well. Changes, like data, can appear in the query body to express complex conditions, like the fact that an entity has been modified in a specific manner, or can be returned by the query in order to retrieve explicit change instances that may have affected specific data. A model that captures both data versions and changes is a prerequisite in order to express such queries, while a query language with specific constructs to support both temporal and change based conditions is needed.

In these terms, in previous work (Stavrakas and Papastefanatos (2010) [55]), a graph model for capturing evolving data and changes, named *evo-graph*, is proposed. In evo-graph changes are complex objects operating on data, exhibiting *structural*, *semantic*, and *temporal* characteristics and they are explicitly modeled as first class citizens distinguised into *basic* and *complex* changes. These properties allow querying evolution on both data and change structure, using temporal- and change-based conditions. Change-centric modelling can provide additional information on *what*, *why*, and *how* data evolved.

On the contrary, several works in literature, like Rizzolo and Vaisman (2008) [49], Gao and Snodgrass (2003) [26], Wang and Zaniolo (2003) [62], that are classified as temporal approaches do not provide any support on querying the changes among data versions, since the notion of changes is not captured explicitly. In other related works, like Marian et al.

(2001) [38] or Chien, Tsotras and Zaniolo (2001) [13], that are classified as version-based approaches change related queries may be supported, but no specific query language is introduced and only a set of basic change operations is considered. Also, the notion of time is not considered.

Additionally, in previous work (Stavrakas and Papastefanatos (2011) [56]) an XML representation for evo-graph, named *evoXML*, is proposed. The Extensible Markup Language (XML) [7] is a simple text-based format for exchanging data on the Web. XML documents are made up of units, named entities, which form a tree structure, and may have attributes and text content. XQuery [52] is the standard query language for querying XML data, building upon XPath [51], a language based on path expressions to navigate through an XML document and select data nodes. The XML, XQuery and XPath are W3C recommendations.

Building upon previous work, we formally define *evo-path*, an XPath (Robie, Dyck and Spiegel (2017) [51]) extension for performing time-aware and change-aware queries on evo-graph. Evo-path allows querying both data history and change structure in a uniform way, taking advantage of changes in order to retrieve and relate data that are otherwise irrelevant to each other. *Temporal*, *evolution* and *causality* queries are supported. Also, we implemented and experimentally evaluated the basic concepts of evo-graph in the *C2D framework*, using XML technologies. The Chapter main contributions are the following:

- formalizing evo-path syntax,
- defining evo-path formal semantics,
- presenting evo-path implementation based on a formal translation of evo-path into equivalent XPath expressions over evoXML,
- evaluating the C2D framework in terms of the space efficiency of evoXML and the performance of the *reduction* process, the process for generating a snapshot holding under a specific time instance from evo-graph.

The Chapter outline is as follows: Section 4.2 presents a motivating example of this work. Section 4.3 presents previous work on evo-graph, evoXML, basic and complex changes. Section 4.4 formally defines evo-path, presenting evo-path syntax, semantics, implementation and illustrative examples. Section 4.5 presents the C2D framework and the evaluation performed.

4.2. Motivating Example

Consider an example taken from Biology, the revision in the classification of diabetes, which was caused by a better understanding of insulin (National research council (2005) [42]). Initially, diabetes was classified according to the age of the patient, as *juvenile* or *adult onset*. As the role of insulin became clearer two more subcategories were added: *insulin dependent* and *non-insulin dependent*. All *juvenile* cases of diabetes are *insulin dependent*, while *adult onset* may be either *insulin dependent* or *non-insulin dependent*.

In Figure 4, the leftmost image depicts a tree representation of the initial diabetes classification, while the rightmost image the revised diabetes classification. Supposing that a scientist examines the revised classification, she may realize that diabetes categories are not as expected. She would like to know:

- *Which may be the previous structure of categories?*
- *Which changes are responsible for the reorganization of diabetes categories?*
- *What are the previous versions of the data nodes that changed due to the reorganization of diabetes categories?*

The first question corresponds to a *temporal* query, on the history of data nodes. The second to an *evolution* query, on the changes applied on data nodes. The third question corresponds to a *causality* query, on the relationships between change nodes and data nodes.

However, these representations are not informative on which parts of the data evolved and how, which changes led from one version to another, or what changes were applied on which parts of data. Recording change operations in a log or computing deltas between successive versions do not solve the problem. As a result, answering such questions may require complex queries in different parts of a database, a task which may be even more intensive for large datasets. The need for tracing past changes and data lineage is evident in a wide range of web information management domains.

The middle image in Figure 4 depicts the representation of the revision in the diabetes classification from the graph of Figure 4 left to right in evo-graph. In evo-graph, both data and changes are uniformly represented: data versions are represented in circular nodes, while changes in triangular nodes, and both are organized in hierarchical structure. Change nodes connect with the data versions they affect and they are annotated with temporal information. As a result, evo-graph may support queries that refer on data versions as well as on changes.

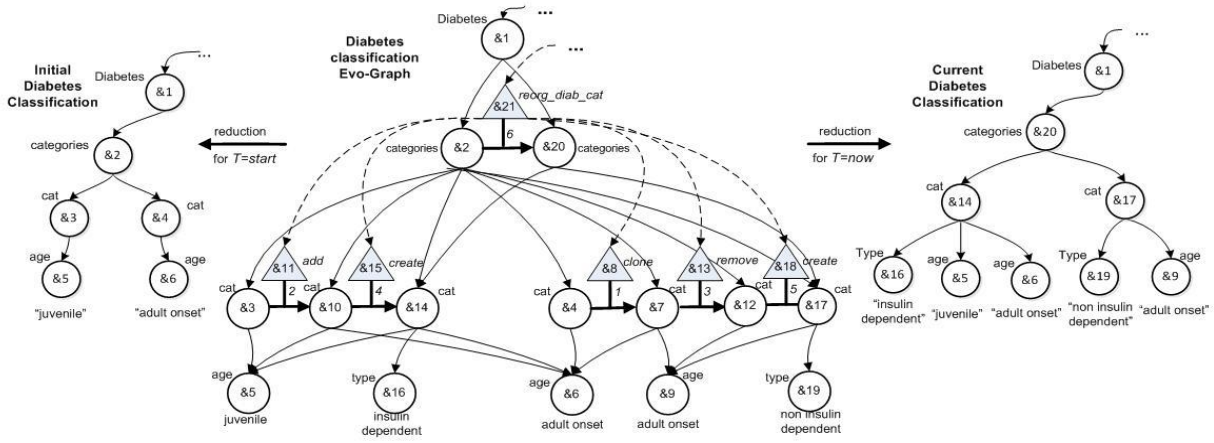


Figure 4 Snap-models of diabetes classification before (left) and after (right) revision and the relevant evo-graph (middle).

4.3. Preliminaries: Modeling Data Versions and Changes on Evo-Graph

Based on Stavrakas and Papastefanatos (2010, 2011) [55] and [56] we present the following preliminary concepts.

Snap-model. In terms of this work, we assume that data is represented by a rooted, node-labeled, leaf-valued tree called snap-model. A snap-model S (V, E) consists of a set of nodes V, divided into complex and atomic, with atomic being the tree leaves, and a set of directed edges E. At any time instance, snap-model undergoes arbitrary changes.

Evo-graph. An *evo-graph* G is a graph-based model that captures all the instances of an evolving snap-model across time, together with the changes responsible for the transitions. It consists of the following components:

- *Data nodes*, divided into *complex* and *atomic*: $V_D = V_D^c \cup V_D^a$.
- *Data edges*, departing from every complex data node, $E_D \subseteq (V_D^c \times V_D)$.
- *Change nodes*, representing change events. They are depicted as triangles to distinguish from circular data nodes. They are divided into *complex* and *atomic* (denoting basic change operations): $V_C = V_C^c \cup V_C^a$.
- *Change edges*, connecting every complex change node to the (complex or atomic) change nodes it contains: $E_C \subseteq (V_C^c \times V_C)$.

- *Evolution edges*, connecting each change node with two data nodes, the version before and after the change: $E_E \subseteq (V_D \times V_C \times V_D)$.
- $r_D \in V_D$ is the *data root*, with the property that there exists a path formed by data edges from r_D to every other data node in V_D.
- $r_C \in V_C$ is the *change root*, with the property that there exists a path formed by change edges from r_C to every other change node in V_C.

Intuitively, evo-graph consists of a data graph, holding the data versions, and a tree of changes, which interconnect via evolution edges. Consequently, it has two roots: the data root, r_D, and the change root, r_C.

Moreover, change nodes are annotated with timestamps denoting the time instance each change occurred. Although valid time may be considered, we rely on transaction time, assuming a linear time domain constituted by consecutive discrete values and two special time instances: 0 for the beginning of time and now for the current time. Also, the timestamp of each complex change equals the timestamp of its most recently occurred child change, since a complex change occurs when all of its constituent changes have been occurred.

In evo-graph, timestamps are used for determining the validity timespan of all data nodes and data edges. Evo-graph can then be reduced to a snap-model holding under a specified time instance through the *reduction* process [55].

As an evo-graph example consider the middle image in Figure 4, representing the revision in the diabetes classification from the graph of Figure 4 left to right. The revision process is denoted by the complex change *reorg_diab_cat* (node &21) composed by 5 basic snap changes (in the order they occurred): *clone* (node &8), *add* (node &11), *remove* (node &13), *create* (node &15), and *create* (node &18). Note the use of evolution edges; in the case of add the evolution edge is annotated with the timestamp 2 and connects node &3 (initial version) with node &10 (version after adding the child node &6). Node &10 is still a child of node &2, but for simplicity the relevant edge is omitted. The reduction of the evo-graph for T=*start* (i.e. 0) results in the snap-model of the leftmost image of Figure 4, while for T=*now* in the snap-model of the rightmost image of Figure 4.

Basic and Complex Changes. The following *basic change operations* may be applied on a snap-model (snap changes for short):

- *create(v^P, v, label, value)*. Creates a new atomic node *v* with a given *label* and *value* and connects it with its parent node v^P. If v^P is an atomic node, it becomes complex.
- *add(v^P, v)*. Adds the edge *(v^P, v)* to E, effectively adding *v* as a child node of v^P. The nodes v^P, *v* must already exist in V. If v^P is an atomic node, it becomes complex.
- *remove(v^P, v)*. Removes the edge *(v^P, v)* from E. If *v* has no other incoming edges, it is removed from V. If v^P has no other children, it becomes an atomic node with the default value (empty string).
- *update(v, newValue)*. Updates the value of an atomic node *v* to *newValue*.
- *clone(v^P, v^{source}, v^{clone})*. Creates a new data node v^{clone} with the same label/value as v^{source}, and a deep copy of the subtree under v^{source} as a subtree under the node v^{clone}. The node v^P must be a parent of v^{source}. The edge (v^P, v^{clone}) is added to E, making v^{clone} a sibling of v^{source}.

The above definitions describe the effect of each snap change to the current snap-model. These changes leave the snap-model in any possible consistent state. Note that the effect of the clone snap-change is to create a deep copy of a subtree under the same parent node. Although clone can be expressed as a sequence of other snap changes, it is chosen to be a basic operation. The reason is that deep copy is difficult to express using successive create operations, while at the same time it is an essential operation for expressing complex changes like move-to, and copy-to.

Figure 5 depicts how each snap change is captured in evo-graph [44]. Figure 5 depicts three images for each snap change: the leftmost shows the initial snap-model before the change, the rightmost shows the current snap-model after the snap change, and the middle image shows the evo-graph fragment encompassing both snapshots, together with the change. Notice that on evo-graph, each snap change evolves the node it applies on into a new version which actually captures its effect.

A *complex change* applied on a node of a snap-model is a sequence of basic and other complex change operations that are applied on the node itself or/and its descendants, formulating semantically coherent sequences. Applying a complex change on a snap-model involves the application of each constituent change in the order they appear.

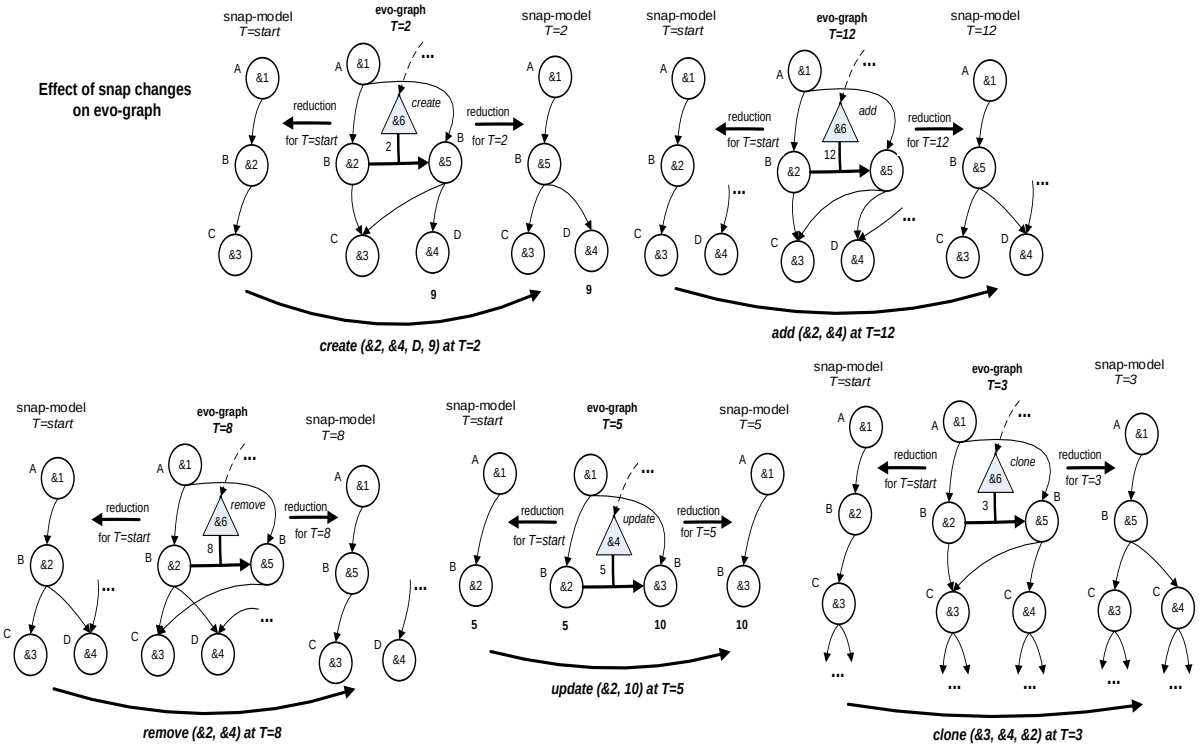


Figure 5 Effect of snap change operations on the evo-graph.

A complex change example is *reorg_diab_cat* applied on categories node of Figure 4 leftmost image. On evo-graph *reorg_diab_cat* evolves node &2 into &20. The definition is given below:

```
reorg-diab-cat (&2) {
 clone (&4, &6, &9)
 add (&3, &6)
 remove (&4, &6)
 create (&3, &16, 'type', 'insulin dependent')
 create (&4, &19, 'type', 'non insulin dependent')
}
```

EvoXML. In Stavrakas and Papastefanatos (2011) [56] an XML representation of evo-graph, named *evoXML*, is presented. evoXML encodes evo-graph in a *top-down non-replicated* approach. Non-replicated means that XML references are used to connect the parent nodes to a common child element. Top-down means that common children are pointed to by their parents via references.

In evoXML special-purpose elements and attributes are used with the namespace `evo`. The evo-graph data root and change root are mapped to the elements `evo:DataRoot` and `evo:ChangeRoot` respectively. Each element is tagged with the label of the respective node and has an attribute `evo:id` whose value is the respective node id in the evo-graph. The values of atomic data nodes are the content of the respective elements, while atomic

Table 15 EvoXML for time instance 1

```
1   <evo:evoXML xmlns=""
2                   xmlns:evo="http://web.imis.athena-innovation.gr/projects/c2d">
3    <evo:DataRoot evo:id="dataroot">
4     <Diabetes evo:id="1" evo:ts="0" evo:te="now">
5      <categories evo:id="2" evo:ts="0" evo:te="now">
6       <cat evo:id="3" evo:ts="0" evo:te="now">
7        <age evo:id="5" evo:ts="0" evo:te="now">
8         juvenile
9        </age>
10      </cat>
11      <cat evo:id="4" evo:ts="0" evo:te="0">
12       <age evo:id="6" evo:ts="0" evo:te="now">
13        adult onset
14       </age>
15      </cat>
16      <cat evo:id="7" evo:ts="1" evo:te="now" evo:previous="4">
17       <age evo:ref="6"/>
18       <age evo:id="9" evo:ts="1" evo:te="now">
19        adult onset
20       </age>
21      </cat>
22     </categories>
23    </Diabetes>
24   </evo:DataRoot>
25   <evo:ChangeRoot evo:id="changeroot">
26    <clone evo:id="8" evo:tt="1" evo:before="4"  evo:after="7"/>
27   </evo:ChangeRoot>
28  </evo:evoXML>
```

change nodes are empty elements. A change edge between two change nodes is captured as the parent-child relationship of the corresponding elements. The same holds for data edges. However, if a child node is pointed to by multiple parent versions, the element corresponding to the child node is contained in the oldest parent element, while subsequent parent versions contain "clone" elements of the child. The "clone" elements are empty elements that point to the "original" child element via the special-purpose attribute `evo:ref`. An evolution edge (v_1, c, v_2) is represented via `evo:before` and `evo:after` attributes on the element corresponding to the change node *c*. They reference the elements that represent v_1 and v_2 respectively. Also, the attribute `evo:previous` is used in the element representing v_2 to reference the element representing v_1. Thus, the previous version of an element is spotted directly without having to refer to the `evo:before` attribute of the corresponding change element. Finally, the attribute `evo:tt` records the timestamp of a change node, and the attributes `evo:ts` and `evo:te` the beginning and the end of the validity timespan of a data node (both are inclusive).

For example, Table 15 (above) presents the evoXML for time instance 1 of the evo-graph in Figure 4, including only the *clone* operation (node &8, lines 16-21, 26). Notice that the edge

from node &7 to node &6 (denoting that &6 remains a child of the next version of &4) is represented via the evoXML reference *evo:ref* in line 17, which points to the element in line 12. Also, notice the change node &8 in line 26. Overall, observe that the XML representation is *additive* with respect to evo-graph operations: as the evo-graph evolves, only additions of new elements are performed in the corresponding evoXML document.

4.4. EvoPath Query Language

4.4.1. Syntax

Similar to XPath, evo-path uses path expressions to move through and select data nodes. In addition, evo-path allows the navigation through change nodes on evo-graph. Consequently, there are two types of path expressions in evo-path: *data path* and *change path expressions*. Also, several predicates are supported to express conditions on evo-graph temporal properties and evolution edges.

Data path expressions start from the data root of evo-graph and return data nodes. Similar to XPath, they are written as a sequence of *location steps* separated by "/" characters and shortcuts can be used as in the two equivalent evo-paths below:

```
/child::A/descendant-or-self::node()/
           child::B/child::*[position()=1]
/A//B/*[1]
```

Change path expressions start from the change root of evo-graph and return change nodes. They have the same syntax as data path expressions, but are enclosed in square brackets:

```
</location_step1/…/location_stepN>
```

Temporal predicates are introduced in evo-path in order to express temporal conditions on the evo-graph nodes. The following types are employed:

1) On data node timespan:

`[ts() operator (t_1, t_2)]`, where `ts()` evaluates to the validity timespan of the context data node, `operator` may be `[not] (in | contains | meets | equals)` covering the standard operations between sets, allowing the use of `not` in front of any of the operators, and `t_1, t_2` are specified timestamps defining a timespan.

`[ts() operator t]`, where `ts()` evaluates to the validity timespan of the context data node, `operator` may be `[not] covers`, and `t` is a specified timestamp, for the case where a specified timestamp exists or not in the validity timespan.

2) On data node timespan start time:

`[tstart() operator t]`, where `tstart()` evaluates to the start of the validity timespan of the context data node, `operator` may be `(> | >= | = | < | <=)`, and `t` is a specified timestamp.

3) On data node timespan end time:

`[tend() operator t]`, where `tend()` evaluates to the end of the validity timespan of the context data node (`operator` and `t` as in case 2).

4) On change node timestamp:

`[tt() operator t]`, where `tt()` evaluates to the timestamp of the context change node (`operator` and `t` as in case 2).

Evolution predicates are used to assert the existence of evolution edges at specific points in the graph. They combine a data path expression with a change path expression and vice versa, implying that the specified data are affected by the specified change. Their general form is:

5) `data_path_expr [evo-filter(<change_path_expr>)]`

6) `<change_path_expr [evo-filter(data_path_expr)]>`

where `evo-filter` may be one of: `evo-before()`, `evo-after()` and `evo-both()`.

Each `evo-filter` evaluates into true or false, in case there is or not an evolution edge involving the data or change node in context. `evo-before()` and `evo-after()` refer on a specific side of the evolution edge, while `evo-both()` on both sides. In case 5 `evo-before()` and `evo-after()` validate whether the data node in context holds before and

after respectively the application of the change being represented by the change node in context. `evo-both()` validates whether the data node holds either before or after the change. In case 6 `evo-before()` and `evo-after()` validate whether the change node in context represents the change before and after which the data node in context holds respectively. `evo-both()` validates whether the change node represents the change either before or after which the data node holds.

4.4.2. Example Queries

The evo-path examples refer to and are evaluated on the evo-graph of Figure 4 regarding diabetes.

1) Temporal queries - Querying the history of data elements: Suppose that a scientist examines the current diabetes snapshot and realizes that the categories structure is not as expected. She wants to retrieve the previous versions of data node &20.

```
//Diabetes/categories[ts() not covers 'now']     (Q1)
```

This is a data path expression with a temporal predicate that evaluates false for the current version of `categories` and true for every other version. It returns node &2 with timespan [0, 5].

2) Evolution queries - Querying changes applied on data elements: The scientist observes the creation of several new nodes under the categories node. She wants to know the complex changes that contain a relevant create operation, to check if create was part of a larger modification.

```
<//* [evo-both(//Diabetes//*)]
     [.//create [evo-both(//Diabetes/categories/cat)]]>    (Q2)
```

This is a change path expression. The first predicate is an evolution predicate for returning all the change nodes that are applied to `Diabetes` node or any of its descendants. The second predicate dictates that only changes with a `create` descendant applied on a `cat` object can be returned. It returns node &21 with timestamp 6, i.e. the complex change *`reorg_diab_cat`*, affecting data node &2 and resulting into data node &20.

The scientist can now retrieve all the changes associated with *reorg_diab_cat*, in order to understand its full effect.

```
<//reorg_diab_cat/*>    (Q3)
```

This change path expression returns the change nodes &8, &11, &13, &15 and &18.

3) Causality queries - Querying relationships between change and data elements: Realizing that the modifications on diabetes categories are related to the complex change &21 *reorg_diab_cat*, the scientist decides to check all the previous versions of the data nodes affected by *reorg_diab_cat* and its descendant changes.

```
//* [evo-before(<//reorg_diab_cat//*>)]    (Q4)
```

The data path expression returns all data nodes being connected through evolution edges with a *reorg_diab_cat* change node (&21) or one of its descendant change nodes, specifically those before each change due to `evo-before()`. The nodes &3 with timespan [0, 1], &4 [0, 0], &7 [1, 2], &10 [2, 3] and &12 [3, 4] are returned. The scientist now realizes the sequence of data evolution.

4.4.3. Semantics

In XPath, the meaning of a path expression is the sequence of nodes, at the end of each path, that matches the expression. In evo-path, the meaning of a data path expression is a sequence of (data-node, interval) pairs such that the data-node has been at the end of a matching data path continuously during that interval. The interval is the validity timespan of the data-node. In evo-path, the meaning of a change path expression is a sequence of (change-node, instance, data-node-before, data-node-after) tuples such that the change-node is at the end of a matching change path at the specific instance and it references the data-node-before and the data-node-after the change. The instance is the timestamp (transaction time) when the change was applied on the data-node-before, leading to the data-node-after.

For specifying the evo-path semantics the formal XPath semantics introduced by Wadler (1999) [60] have been adapted. The meaning of an XPath expression is specified with respect to a context node. For a data path expression, this is extended to a context pair of a data-node and a time interval. For a change path expression, its meaning is specified with respect to a

context tuple of a change-node, a time instance, a data-node before and data-node after the change. For the data part, four semantic functions are defined: S, Q, Q_T and Q_E. $S[\![p]\!]x$ denotes the sequence of pairs (data-node, interval) selected by pattern p when x is the context pair. It may also denote a sequence of values. The boolean expression $Q[\![q]\!]x$ denotes whether or not the qualifier q is satisfied when the context pair (data-node, interval) is x. The boolean expression $Q_T[\![q_T]\!]x$ denotes whether or not a temporal condition q_T is satisfied, while the boolean expression $Q_E[\![q_E]\!]x$ denotes whether or not an evolution condition q_E is satisfied.

For the change part, four similar semantic functions are defined: S_c, Q_c, Q_{cT} and Q_{cE}. $S_c[\![p]\!]x$ denotes the sequence of tuples (change-node, instance, data-node-before, data-node-after) selected by pattern p when x is the context tuple. It may also denote a sequence of values. The boolean expression $Q_c[\![q]\!]x$ denotes whether or not the qualifier q is satisfied when the context tuple (change-node, instance, data-node-before, data-node-after) is x. The boolean expression $Q_{cT}[\![q_T]\!]x$ denotes whether or not a temporal condition q_T is satisfied, while the boolean expression $Q_{cE}[\![q_E]\!]x$ denotes whether or not an evolution condition q_E is satisfied. In Table 16 the formal semantics of the most common evo-path constructs are presented.

For the *data root* and *change root* it holds: The validity timespan of the *data root* is by definition $[0, \text{now}]$, as it is always valid in time. The timestamp of the *change root* is by definition 0, the data-node-before and data-node-after are undefined ($\emptyset$), as it does not represent an actual change.

Table 16 Formal Semantics of Evo-Path

$$S[\![/p]\!]x = S[\![p]\!]dataRoot(x);$$
$$S[\![//p]\!]x = \{x_2 | x_1 \in subnodes(dataRoot(x)), x_2 \in S[\![p]\!]x_1\};$$
$$S[\![p_1/p_2]\!]x = \{(v_2, I_1 \cap I_2) | (v_1, I_1) \in S[\![p_1]\!]x, (v_2, I_2) \in S[\![p_2]\!](v_1, I_1)\};$$
$$S[\![p_1//p_2]\!]x = \{x_3 | x_1 \in S[\![p_1]\!]x, x_2 \in subnodes(x_1), x_3 \in S[\![p_2]\!]x_2\};$$
$$S[\![p[q]]\!]x = \{(v, I) | (v, I) \in S[\![p]\!]x, Q[\![q]\!](v, I)\};$$
$$S[\![n]\!]x = \{(v, I) | isElement(v), child(x) = (v, I), name(v) = n\};$$
$$S[\![tstart()]\!]x = \{s | x = (v, I), I = [s, e]\};$$
$$S[\![tend()]\!]x = \{e | x = (v, I), I = [s, e]\};$$
$$S[\![p[q_T]]\!]x = \{(v, I) | (v, I) \in S[\![p]\!]x, Q_T[\![q_T]\!](v, I)\};$$
$$S[\![ancestor :: p]\!]x = \{x_2 | x_1 \in prenodes(x), x_2 \in S[\![p]\!]x_1\};$$
$$Q[\![p = s]\!]x = \{(v, I) | (v, I) \in S[\![p]\!]x, value(v) = s\} \neq \emptyset;$$
$$Q[\![p]\!]x = \{x_1 | x_1 \in S[\![p]\!]x\} \neq \emptyset;$$
$$Q_T[\![ts()\ in\ (t_1, t_2)]\!]x = \{x | x = (v, [t_{start}, t_{end}]), t_{start} \geq t_1, t_{end} \leq t_2\} \neq \emptyset;$$
$$Q_T[\![ts()\ contains\ (t_1, t_2)]\!]x = \{x | x = (v, [t_{start}, t_{end}]), t_{start} \leq t_1, t_{end} \geq t_2\} \neq \emptyset;$$
$$Q_T[\![ts()\ meets\ (t_1, t_2)]\!]x = \{x | x = (v, [t_{start}, t_{end}]), [t_{start}, t_{end}] \cap [t_1, t_2] \neq \emptyset\} \neq \emptyset;$$
$$Q_T[\![ts()\ equals\ (t_1, t_2)]\!]x = \{x | x = (v, [t_{start}, t_{end}]), t_{start} = t_1, t_{end} = t_2\} \neq \emptyset;$$
$$Q_T[\![ts()\ covers\ t]\!]x = \{x | x = (v, [t_{start}, t_{end}]), t \geq t_{start}, t \leq t_{end}\} \neq \emptyset;$$
$$Q_T[\![tstart()\ operator\ t]\!]x = \{x | x = (v, [t_{start}, t_{end}]), t_{start}\ operator\ t\} \neq \emptyset;$$
$$Q_T[\![tend()\ operator\ t]\!]x = \{x | x = (v, [t_{start}, t_{end}]), t_{end}\ operator\ t\} \neq \emptyset;$$
$$Q_E[\![evo - before(\langle change\_path\_expr \rangle)]\!]x =$$
$$\{x | x = (v, I), (v_c, i, v_b, v_a) \in S_c[\![\langle change\_path\_expr \rangle]\!]r_c, v = v_b\} \neq \emptyset;$$

$Q_E[\![evo-after(\langle change\_path\_expr\rangle)]\!]x =$
$\{x|x=(v,I),(v_c,i,v_b,v_a)\in S_c[\![\langle change\_path\_expr\rangle]\!]r_c, v=v_a\}\neq\emptyset;$
$Q_E[\![evo-both(\langle change\_path\_expr\rangle)]\!]x =$
$\{x|x=(v,I),(v_c,i,v_b,v_a)\in S_c[\![\langle change\_path\_expr\rangle]\!]r_c, v=v_a \vee v=v_b\}\neq\emptyset;$
$S_c[\![\langle/p\rangle]\!]x = S_c[\![p]\!]changeRoot(x);$
$S_c[\![\langle//p\rangle]\!]x = \{x_2|x_1\in subnodes_c(changeRoot(x)), x_2\in S_c[\![p]\!]x_1\};$
$S_c[\![\langle p_1/p_2\rangle]\!]x = \{x_2|x_1\in S_c[\![p_1]\!]x, x_2\in S_c[\![p_2]\!]x_1\};$
$S_c[\![\langle p_1//p_2\rangle]\!]x = \{x_3|x_1\in S_c[\![p_1]\!]x, x_2\in subnodes_c(x_1), x_3\in S_c[\![p_2]\!]x_2\};$
$S_c[\![\langle p[q]\rangle]\!]x = \{(v_c,i,v_b,v_a)|(v_c,i,v_b,v_a)\in S_c[\![p]\!]x, Q_c[\![q]\!](v_c,i,v_b,v_a)\};$
$S_c[\![n]\!]x = \{(v_c,i,v_b,v_a)|isElement(v_c), child_c(x)=(v_c,i,v_b,v_a), name(v_c)=n\};$
$S_c[\![tt()]\!]x = \{i|x=(v_c,i,v_b,v_a)\};$
$S_c[\![\langle p[q_T]\rangle]\!]x = \{(v_c,i,v_b,v_a)|(v_c,i,v_b,v_a)\in S_c[\![p]\!]x, Q_{cT}[\![q_T]\!](v_c,i,v_b,v_a)\};$
$S_c[\![\langle ancestor :: p\rangle]\!]x = \{x_2|x_1\in prenodes_c(x), x_2\in S_c[\![p]\!]x_1\};$
$Q_c[\![p=s]\!]x = \{(v_c,i,v_b,v_a)|(v_c,i,v_b,v_a)\in S_c[\![p]\!]x, value(v)=s\}\neq\emptyset;$
$Q_c[\![p]\!]x = \{x_1|x_1\in S_c[\![p]\!]x\}\neq\emptyset;$
$Q_{cT}[\![tt(\,)\ operator\ t]\!]x = \{x|x=(v_c,i,v_b,v_a), i\ operator\ t\}\neq\emptyset;$
$Q_{cE}[\![evo-before(data\_path\_expr)]\!]x =$
$\{x|x=(v_c,i,v_b,v_a),(v,I)\in S[\![data\_path\_expr]\!]r_d, v=v_b\}\neq\emptyset;$
$Q_{cE}[\![evo-after(data\_path\_expr)]\!]x =$
$\{x|x=(v_c,i,v_b,v_a),(v,I)\in S[\![data\_path\_expr]\!]r_d, v=v_a\}\neq\emptyset;$
$Q_{cE}[\![evo-both(data\_path\_expr)]\!]x =$
$\{x|x=(v_c,i,v_b,v_a),(v,I)\in S[\![data\_path\_expr]\!]r_d, v=v_a \vee v=v_b\}\neq\emptyset;$

Where:
$subnodes(y) = \{(v,I)|there\ exists\ a\ data\ path\ from\ y\ to\ v\ and\ I\ is\ the\ validity\ timespan\ of\ v\}$
$prenodes(y) =$
$\{(v,I)|there\ exists\ a\ data\ path\ from\ v\ to\ y\ and\ I\ is\ the\ validity\ timespan\ of\ v\},$
$dataRoot(x)$ *is the* $(dataRoot,[0,now])$ *pair where* $dataRoot$ *is the root of the graph in which* $data-node$ *exists and* x *is a* $(data-node, interval)$ *pair,* $r_d=(dataRoot,[0,now])$,
$child(x) =$
$\{(v,I)|there\ exists\ a\ data\ path\ of\ length\ 1\ from\ x\ to\ v\ and\ I\ is\ the\ validity\ timespan\ of\ v\}$
$subnodes_c(y) =$
$\{(v_c,i,v_b,v_a)|there\ exists\ a\ change\ path\ from\ y\ to\ v_c\ and\ i\ is\ the\ timestamp\ of\ v_c\}$
$prenodes_c(y) =$
$\{(v_c,i,v_b,v_a)|there\ exists\ a\ change\ path\ from\ v_c\ to\ y\ and\ i\ is\ the\ validity\ timespan\ of\ v_c\},$
$changeRoot(x)$ *is the* $(changeRoot,0,\emptyset,\emptyset)$ *tuple where* $changeRoot$ *is the root of the graph in which* $change-node$ *exists and* x *is a* $(change-node, instance, data-node-before, data-node-after)$ *tuple,* $r_c=(changeRoot,0,\emptyset,\emptyset)$,
$child_c(x) =$
$\{(v_c,i,v_b,v_a)|there\ exists\ a\ change\ path\ of\ length\ 1\ from\ x\ to\ v_c\ and\ i\ is\ the\ timestamp\ of\ v_c\}$

4.4.4. Implementation

In order to implement evo-path, each valid evo-path expression is translated into an equivalent XPath expression over evoXML. Table 17 summarizes the translation rules.

Table 17 Evo-Path to XPath translation

Evo-Path	XPath
A. Data and Change Path Expressions	
data_path_expr	doc("evoXML.xml")/evo:evoXML/evo:DataRoot/mapped_data_path_expr
<change_path_expr>	doc("evoXML.xml")/evo:evoXML/evo:ChangeRoot/mapped_change_path_expr

B. Temporal Predicates	
[ts() in (t_1, t_2)], where $t\_2 \in \mathbb{N}$	[@evo:ts>= t_1 and (if @evo:te='now' then false() else @evo:te<= t_2)]
[ts() contains (t_1, t_2)], where $t\_2 \in \mathbb{N}$	[@evo:ts<= t_1 and (if @evo:te='now' then true() else @evo:te>= t_2)]
[ts() meets (t_1, t_2)], where $t\_2 \in \mathbb{N}$	[if @evo:te='now' then (@evo:ts>= t_1 and @evo:ts<= t_2) else((@evo:ts>= t_1 and @evo:ts<= t_2) or (@evo:te>= t_1 and @evo:te<= t_2))]
[ts() equals (t_1, t_2)], where $t\_2 \in \mathbb{N}$	[@evo:ts = t_1 and (if @evo:te='now' then false() else @evo:te = t_2)]
[ts() in (t_1, 'now')]	[@evo:ts>= t_1]
[ts() contains (t_1, 'now')]	[@evo:ts<=t_1 and @evo:te='now']
[ts() meets (t_1, 'now')]	[if @evo:te='now' then true() else (@evo:ts>=t_1 or @evo:te>=t_1)]
[ts() equals (t_1, 'now')]	[@evo:ts = t_1 and @evo:te='now']
[ts() covers t], where $t \in \mathbb{N}$	[@evo:ts<= t and (if @evo:te='now' then true() else @evo:te>= t)]
[ts() covers 'now']	[@evo:te='now']
[tstart() operator t], where $t \in \mathbb{N}$	[@evo:ts operator t]
[tend() > t], where $t \in \mathbb{N}$	[if @evo:te='now' then true() else @evo:te> t]
[tend() >= t], where $t \in \mathbb{N}$	[if @evo:te='now' then true() else @evo:te>= t]
[tend() = t], where $t \in \mathbb{N}$	[if @evo:te='now' then false() else @evo:te = t]
[tend() < t], where $t \in \mathbb{N}$	[if @evo:te='now' then false() else @evo:te< t]
[tend() <= t], where $t \in \mathbb{N}$	[if @evo:te='now' then false() else @evo:te<= t]
[tend() = 'now']	[@evo:te='now']
[tend()< 'now']	[@evo:te!='now']
[tend()<= 'now']	[true()]
[tt() operator t], where $t \in \mathbb{N}$	[@evo:tt operator t]
C. Evolution Predicates	
data_path_expr [evo-before(<change_path_expr>)]	doc("evoXML.xml")/evo:evoXML/evo:DataRoot/data_path_expr[@evo:id= doc("evoXML.xml")/evo:evoXML/evo:ChangeRoot/change_path_expr/@evo:before]
data_path_expr [evo-after(<change_path_expr>)]	doc("evoXML.xml")/evo:evoXML/evo:DataRoot/data_path_expr[@evo:id= doc("evoXML.xml")/evo:evoXML/evo:ChangeRoot/change_path_expr/@evo:after]
data_path_expr [evo-both(<change_path_expr>)]	doc("evoXML.xml")/evo:evoXML/evo:DataRoot/data_path_expr[@evo:id= doc("evoXML.xml")/evo:evoXML/evo:ChangeRoot/change_path_expr/@evo:before or @evo:id= doc("evoXML.xml")/evo:evoXML/evo:ChangeRoot/change_path_expr/@evo:after]
<change_path_expr [evo-filter(data_path_expr)]>where evo-filter is evo-before or evo-after or evo-both are defined symmetrically	
D. Plain Data Path Expressions	
1 /p	/p[@evo:id]
2 /p[position predicate]	/p[(@evo:id and position predicate) or (@evo:id=/p[position predicate]/@evo:ref)]
3 /p1[p2 op value]	/p1[@evo:id and p2 op value] | /p1[@evo:id and p2/@evo:ref=/p1[p2 op value]/p2/@evo:id]
4 /p1[p2 op value]/p3	(/p1[@evo:id and p2 op value] | /p1[@evo:id and p2/@evo:ref=/p1[p2 op value]/p2/@evo:id] | /p1[p3/@evo:id=/p1[p2 op value]/p3/@evo:ref])/p3[@evo:id]

Each data/change path expression (case A) is evaluated starting from the *data/change root*. Each temporal predicate (case B) is mapped to an XPath predicate over evoXML attributes `evo:ts`, `evo:te` and `evo:tt`. Each evolution predicate (case C) is mapped to an XPath predicate over the evoXML attributes `evo:before` or/and `evo:after`. These attributes appear on change elements and should be equal to `evo:id` attribute of data elements. Moreover, recall that evoXML encodes evo-graph in a top-down non-replicated approach (Stavrakas and Papastefanatos (2011) [56]): if a child node is pointed to by multiple parent versions, the element corresponding to the child node is contained in the oldest parent element, while subsequent parent versions contain "clone" elements of the child. These are empty elements pointing to the "original" child element via `evo:ref` attribute. This feature is handled while translating a data path expression to an equivalent XPath expression (case D). The returned nodes of a data path expression should be the "original" ones, i.e. those with an `evo:id` attribute (rule 1). Similar holds for predicates that are used to find a specific node, e.g. based on position (rule 2). For predicates that are used to find a node that contains a specific value, the returned nodes should be the "original" ones and the contained value should be checked in an "original" child node. However, the node in context may have either an "original" or a "clone" child node. In the latter case, the "clone" child node is used to access the pointed "original" one. Thus, in rule 3 two cases are identified: `p1` is an "original" node and contains the "original" node `p2` with value, or `p1` is an "original" node and contains the "clone" node `p2` pointing to an "original" node with value. This is extended in rule 4 with an additional location step. For `p3` a third case is identified: `p1` is an "original" node which contains the "original" node `p2` with value and the "clone" node `p3`, which is used to access the "original" pointed node `p3`. The case of having `p1` as "original" node and `p2` and `p3` as "clone" nodes is not identified, since it eventually ends up to one of the rest cases. Finally, note that XPath predicates on other node types, like attributes, are not considered, since in evoXML evolving data are represented on element nodes.

Below, we show the XPath expressions for the Section 4.4.2 evo-path queries, generated following the translation rules. For simplicity evo namespace is omitted. evoXML.xml contains the evoXML representation of evo-graph in Figure 4.

```
(Q1)  let $d:=doc("evoXML.xml")/evo:evoXML/evo:DataRoot
      return $d//Diabetes/categories[@evo:te!='now']
(Q2)  let $d:=doc("evoXML.xml")/evo:evoXML/evo:DataRoot,
          $c:=doc("evoXML.xml")/evo:evoXML/evo:ChangeRoot
      return $c//*[@evo:before=$d//Diabetes//*/@evo:id or
                   @evo:after=$d//Diabetes//*/@evo:id]
```

```
        [ .//evo:create[@evo:before=
      $d//Diabetes/categories/cat/@evo:id or
                        @evo:after=
      $d//Diabetes/categories/cat/@evo:id] ]
(Q3)  let $c:=doc("evoXML.xml")/evo:evoXML/evo:ChangeRoot
      return $c//reorg_diab_cat/*
(Q4)  let $d:=doc("evoXML.xml")/evo:evoXML/evo:DataRoot,
          $c:=doc("evoXML.xml")/evo:evoXML/evo:ChangeRoot
      return $d//*[@evo:id=$c//reorg_diab_cat//*/@evo:before]
```

4.5. Evaluating the C2D Framework

4.5.1. The C2D Framework

The C2D (standing for Complex Changes in Data evolution) framework captures the concepts presented on evo-graph, snap-model and evo-path using XML technologies. Currently, the basic concepts of evo-graph and snap-model have been implemented into the framework. C2D has been developed in Java, on top of Berkeley DB XML[9], an embedded XML database used to manage the evoXML representation of evo-graphs.

The basic flow implemented in C2D is the following: Changes applied on the snap-model are fed into a process that populates the evo-graph, which is constructed step-by-step as changes are accommodated on it. The process details are presented in [44]. A snap change is always applied on the current snap-model, which is also represented in XML in C2D. Note that, the current snap-model is actually produced as a reduction of the evo-graph for the time instance T=*now*. In C2D the reduction process, as presented in [55], is also implemented. This flow is depicted in Figure 6. The top layer in Figure 6 is the view layer, where changes are launched. The purpose of the logical model layer is to guide the translation processes between the view layer and the storage representation layer, where changes actually take place. Change operations on the evo-graph are implemented as XML update operations on the corresponding evoXML, using XQuery Update [50] insert expressions.

[9] http://www.oracle.com/technetwork/ database/berkeleydb/overview/index.html

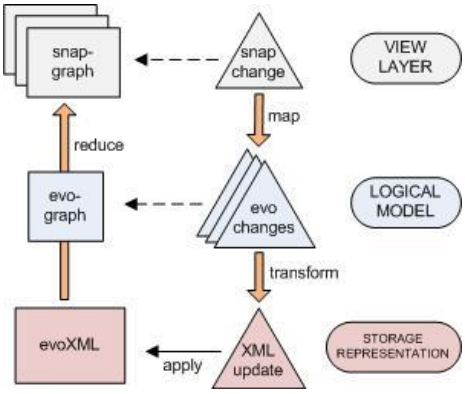


Figure 6 C2D framework basic flow overview.

4.5.2. Evaluation

4.5.2.1. Experimental Setting

The goal of the experimental evaluation was to examine how evo-graph depends on a number of factors that characterize the data. We first examined the space efficiency of evoXML for various configurations, regarding: the structure of the initial XML tree, the type of snap changes, and the selectivity of the elements. We also examined the performance of the reduction process with respect to the size of the evoXML file.

Experiments were performed over synthetic XML data, on a PC with Intel Core 2 CPU 2.26 GHz, and 4.00 GB of RAM. The initial XML file was generated with Xmlgener[10] synthetic XML data generator and contained about 10^5 elements, over which 10^4 snap changes were sequentially applied as XQuery Update [50] statements. A new version was assumed after every 1000 changes; therefore 10 successive versions have been created for each setting. We recorded the size (in terms of the number of XML elements) of each "snap" version, and the size of the evoXML file at the same instance. Furthermore, we examined the performance of the reduction process for the current snapshot (T=now), and the initial snapshot (T=0).

Regarding the structure of the initial data, we used two XML files with the same number of elements: (a) one corresponding to a snap-model with a "deep" tree structure (denoted s_1) with five levels and elements having a fan-out of 10, and (b) a file with a "broad" tree structure (denoted s_2) with only two levels and elements with a fan-out of about 330 elements. We have applied three sets of snap changes: (a) equal percentage for all changes except *clone*

[10] https://code.google.com/archive/p/xmlgener/

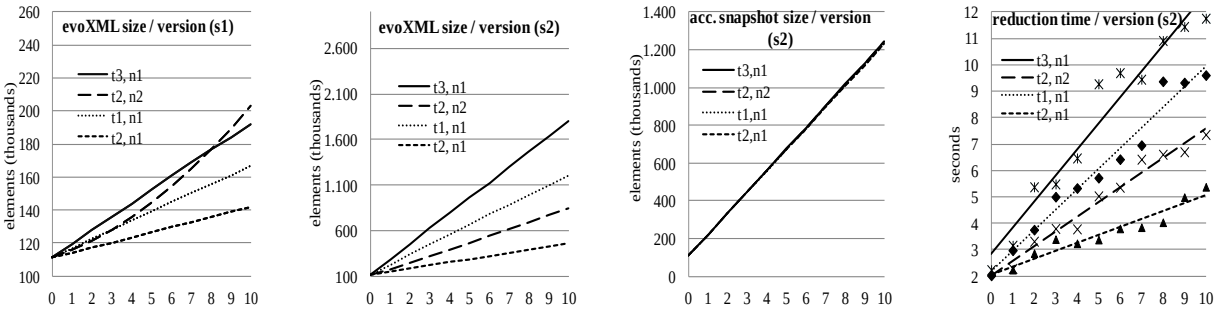
(denoted t_1), (b) 80% *update* and 20% *create* and *remove* (denoted t_2), and (c) equal percentage for all changes including *clone* (denoted t_3). Finally, concerning elements selectivity, changes have been applied either on all elements (denoted n_1) or on a fixed set of pre-selected elements so that each element is affected by 5 changes on average per version (denoted n_2).

We have examined the following combinations of the above parameters: (t_1n_1), (t_3n_1), (t_2n_1), and (t_2n_2) for each of s_1, s_2. t_1n_1 captures the typical case when random changes are uniformly applied on all elements. t_3n_1 is similar to t_1n_1, but it also includes *clone*. We have separately examined the *clone* operation, as it may arbitrarily result in the addition of a large amount of data. t_2n_1 captures the case where most (80%) change operations are *update* on random leaf elements, and only 20% are *create* or *remove*. Finally, t_2n_2 is like the previous case except that changes are concentrated on a pre-selected fixed set of elements.

Intuitively, we expect that the size of the evoXML depends on the number of snap changes performed. We also expect that it depends on the average fan-out of the snap-model, while it remains insensitive to its average height. This is due to the way that each snap change operation is implemented on the evo-graph.

4.5.2.2. Results

In Figure 7 (a) and (b) we present the evoXML sizes per version. Subsequently, we discuss how this size is affected by the aforementioned configurations parameters.



(a) (b) (c) (d)

Figure 7 evoXML size (a), (b), accumulative snapshot size (c) and current snapshot reduction time (d) per version for various configurations.

File structure. For all configurations, better space efficiency is achieved for s_1. For smaller fan-outs (s_1), the evoXML has a smoother increase in size than for large fan-outs (s_2). A snap

change occurring on an element adds *evo:ref* elements for all of its children (i.e. fan-out) that are still valid in the new version. Hence, the increase in the evoXML size is relative to the average fan-out.

Type of changes. t_2 outperforms t_1 and t_3. The majority of changes in t_2 are *update*, which have a smaller impact on the evoXML size. Again, the key point is the number of new elements that each change adds. Observe from Figure 5 that all changes add at least two new elements: one evolved data element and one change element. *update* adds only these two elements, whereas *create* and *add* insert one additional element for the new child, plus *evo:ref* elements for its siblings. *remove* results in inserting *evo:ref* elements in the evoXML for all the siblings of the removed element. Finally, *clone* adds a variable number of elements according to the height and average fan-out of the subtree that is cloned. On the other hand, the percentage of *create* and *remove* in t_1 is higher. In t_3, the use of *clone* further increases the file size by creating a deep copy of the subtree of the elements on which it is applied.

Selectivity of elements. Applying changes randomly on all elements (n_1) seems to have a smoother impact on the increase of the file size (e.g., compare t_2n_1 and t_2n_2 for each of s_1, s_2). This is due to the fact that changes are uniformly distributed over all the elements. On the other hand, the increase is higher when changes are targeting a fixed set of elements (n_2). Changes in t_2n_2 are sequentially applied on the same elements, i.e., create is applied on the same elements, increasing the number of their children and thus the number of *evo:ref* elements to be inserted when a subsequent create occurs on the same element.

Overall, the evoXML size depends almost linearly on the number of the snap changes applied, given that the average fan-out is constant. Moreover, the increase rate of the evoXML size is proportional to the average fan-out of its elements. This is more evident in t_2n_2 for s_1, where the average fan-out of the elements sustaining changes increases significantly per version, resulting in a boost in the evoXML size, whereas in s_2 the fan out increase rate is much smoother.

In Figure 7 (c) we present the accumulative size of the snapshots produced per version. This approach can be considered as an alternative to evoXML. We only depict the series for s_2, as s_1 shows a similar trend. The accumulative size of all snapshots per version is significantly bigger than the evoXML size, for all runs over s_1. The same holds for all configurations of s_2, except for t_3n_1 where many *evo:ref* elements are added in the evoXML file. Note that the overlap of the series is due to the small variance in the accumulative snapshot size between configurations.

Regarding the performance of the reduction algorithm, we have measured the time the reduction process takes for producing the current and the initial snapshots. The results for the current snapshot for s_2 are shown in Figure 7 (d), where the mark signs are the recorded time values, and the series are the trends for each configuration. A safe conclusion is that the reduction time depends mostly on the evoXML size. For small file sizes, the reduction performs the same for all versions. In addition, the increase rates in time are similar for both the current and the initial snapshot, for both s_1 and s_2. Therefore, the time instance parameter of the reduction process does not affect the reduction performance.

Concluding, both space and time efficiency are mostly affected by the average fan-out, which deteriorates as more changes are applied. That is mainly because of the *evo:ref* elements that are added for all children of an element that "evolves". Still, our approach is much more efficient than retaining separately every different version.

Chapter 5

Conclusions and Future Work

5.1. Thesis Conclusions

In this thesis, we have presented novel methods and experimental results that focus on modeling, defining, detecting and querying changes on web data. In the proposed approaches, changes are treated as first class citizens, meaning that they are human-readable, semantically rich changes that demonstrate structure. Therefore, they can play a dominant role in interpreting and understanding evolution.

Based on these concepts our research has been conducted in two pillars: Modeling, defining and detecting changes has been studied in the context of RDF(S) knowledge bases. Querying changes has been studied in the context of XML data, building upon previous work done regarding evo-graph, a model that captures evolving data along with changes.

Specifically, we proposed modeling and supporting simple and complex changes, as well as any relations among them, for interpreting evolution on RDF(S) knowledge-bases. Simple changes are fine-grained and application/data-agnostic changes, while complex changes are coarse-grained and application/data-specific changes, demonstrating structure and rich semantics suitable to each specific application or dataset. Complex changes are user defined changes so that they can capture application/data-specific modifications. Also, they are defined as patterns over simple changes and already defined complex changes. Towards this direction, we proposed an intuitive, user-friendly language, based on change semantics for defining complex changes. We formally defined the language syntax and semantics. Furthermore, the ultimate goal for defining complex changes is to identify actual complex change instances between dataset versions. Therefore, we presented a detection algorithm for the proposed complex change definition language, as well as the correctness of the proposed implementation with respect to the language semantics.

The proposed approach has been extensively evaluated qualitatively and experimentally. The qualitative evaluation demonstrates the added value of our approach in comparison to the related work, regarding the basic features and characteristics. The experimental evaluation examines the complex change language expressiveness and the detection performance. It is evaluated whether the proposed structures are adequate in expressing useful changes and how complex changes facilitate user in understanding and analyzing evolution. Also, the response time of the detection process is examined in terms of increasing dataset size. The evaluation is performed over both artificial and real data, proving the effectiveness of our approach.

Overall, the proposed approach offers expressiveness and flexibility in terms of evolution interpretation, since complex changes provide additional information for interpreting past data, allow interpreting evolution in multiple ways, while capturing relations between complex changes is an additional feature that enriches the complex changes' expressivity.

Regarding querying changes, we formally defined *evo-path*, an XPath extension for performing time-aware and change-aware queries on evo-graph. Evo-path allows querying both data history and change structure in a uniform way, supporting *temporal*, *evolution* and *causality* queries. We presented the evo-path syntax, we defined evo-path formal semantics and we presented an implementation based on a formal translation of evo-path into equivalent XPath expressions over evoXML, the XML representation of evo-graph.

Additionaly, the basic concepts of evo-graph were implemented in the *C2D framework*, using XML technologies, and experimentally evaluated. The space efficiency of evoXML for various configurations is evaluated. Also, the performance of the *reduction* process, the process for generating a snapshot holding under a specific time instance from evo-graph, is evaluated with respect to the size of the evoXML file. The evaluation performed indicated which factors that characterize the data affect the evoXML size and the reduction process.

5.2. Future Work

While conducting the above research, apart from the contributions made and the results already presented, we came up with open issues, which can form the basis for future work.

Regarding our work on defining and detecting complex changes on RDF(S) knowledge bases, a tool for the automatic generation of a proposed set of complex change definitions may be

investigated. This tool may further facilitate the process of defining complex changes over a specific dataset.

Towards this direction, a naïve approach may be to define some common patterns of changes that may appear in any RDF(S) dataset. Thus, given a specific dataset schema, a number of complex changes may be defined automatically following specific rules and heuristics. The proposed complex changes can be named based on the dataset concepts and the intuition of each change. These complex changes may involve rather structural groupings and may model modifications like add, delete, update and move. Therefore, the proposed complex changes may be additions/deletions/updates of class or property instances, moves of property instances, or groupings of added/deleted classes with relevant added/deleted descriptive properties. The data curator or consumer can model new complex changes capturing scenarios and semantics that fit specific data and application use cases on top of them.

Additionally, it may be worth investigating how the automatic generation and proposal of complex change definitions over dataset versions can be based on more advanced methods and procedures than rules or heuristics. This may involve mining structures by comparing and analyzing different versions and snapshots, or ideally identifying unexpected changes.

In this regard, recent works on data structure evolution may be useful. More broadly, our work can be related to approaches capable of capturing the evolution of knowledge graphs, while not aiming to model changes or interpret evolution. In Maillot and Bobed (2018) [37], structural similarity measure is proposed. It is based on the detection of common structural regularities between two RDF graphs, leveraging the data mining approach KRIMP. Bobed et al. (2020) [5] rely on this work, focusing on a data-driven assessment of structural evolution in RDF graphs. They propose two new similarity measures, which identify outdated updates and updates that alter the heterogeneity of the structural patterns w.r.t. the last snapshot. In Gonzalez and Hogan (2018) [28], authors propose an approach to compute a data-driven schema from knowledge graphs, inspired from formal concept analysis (FCA), producing a lattice of characteristic sets. The extracted schema is used to summarize dataset dynamics and predict future changes.

Regarding our work on querying data versions and change structures via evo-path on evo-graph, it is worth focusing on further experimenting and evaluating the proposed approach in terms of query language expressiveness and implementation efficiency. In addition, experimenting on real data may contribute in evaluating the effectiveness of the approach.

Also, another research direction is to investigate prospective optimizations. Towards this direction, it may be useful to take into consideration the effect of evo:ref elements in the evoXML and consequently in the query translation. It may be interesting to work on encoding evo:ref elements and overall compress the evoXML file.

Overall, evolution management may be considered as a special case of the data integration and exchange problems [18], where the involved models are different versions of the same dataset. Therefore, several formalization issues that appear in data integration and exchange, like information preservation, query preservation, monotonicy and containment, can be examined in the evolution context as well.

Bibliography

[1] Amagasa, T., Yoshikawa, M., Uemura, S. (2000). A data model for temporal XML documents. In DEXA.

[2] Antoniazzi, F., and Viola, F. (2018). RDF graph visualization tools: A survey. In 23rd Conference of Open Innovations Association (FRUCT).

[3] Auer, S., and Herre, H. (2007). A versioning and evolution framework for RDF knowledge bases. In Perspectives of Systems Informatics.

[4] Berners-Lee, T., Connolly, D. (2004). Delta: An ontology for the distribution of differences between RDF graphs. http://www.w3.org/DesignIssues/Diff (version: 2006-05-12).

[5] Bobed, C., Maillot, P., Cellier, P., Ferré, S. (2020). Data-driven assessment of structural evolution of RDF graphs. In Semantic Web Journal 11(5): 831-853.

[6] Brahmia, Z., Hamrouni, H., Bouaziz, R. (2020). XML data manipulation in conventional and temporal XML databases: A survey. In Computer Science Review Journal 36: 100231.

[7] Bray, T., Paoli, J., Sperberg-McQueen, C. M., Maler, E., Yergeau, F. (2008, November 26). Extensible Markup Language (XML) 1.0. W3C Recommendation. https://www.w3.org/TR/xml/. (accessed 3 October 2021)

[8] Brickley, D., Guha, R. V. (2014, February 25). RDF Schema 1.1. W3C Recommendation. https://www.w3.org/TR/rdf-schema/. (accessed 3 October 2021)

[9] Buneman, P., Chapman, A.P., Cheney, J. (2006). Provenance management in curated databases. In SIGMOD.

[10] Buneman, P., Khanna, S., Tajima, K., Tan, W.C. (2004). Archiving scientific data. In ACM Transactions on Database Systems 29(1): 2-42.

[11] Chawathe, S., Abiteboul, S., Widom, J. (1999). Managing historical semistructured data. In Theory and Practice of Object Systems 5(3): 143-162.

[12] Chawathe, S., Rajaraman, A., Garcia-Molina, H., Widom J. (1996). Change detection in hierarchically structured information. In SIGMOD.

[13] Chien, S-Y., Tsotras, V. J., Zaniolo, C. (2001). Efficient management of multiversion documents by object referencing. In VLDB.

[14] Chien, S. Y., Tsotras, V. J., Zaniolo, C. (2002). Efficient schemes for managing multiversion XML document. In VLDB Journal 11(4): 332-353.

[15] Chien, S-Y., Tsotras, V. J., Zaniolo, C., Zhang, D. (2001). Storing and querying multiversion XML documents using durable node numbers. In WISE.

[16] Chien, S. Y., Tsotras, V. J., Zaniolo, C., Zhang, D. (2002). Efficient complex query support for multiversion XML documents. In EDBT.

[17] Cobena, G., Abiteboul, S., Marian, A. (2002). Detecting changes in XML documents. In ICDE.

[18] Doan, A., Halevy, A., Ives, Z. (2012). Principles of Data Integration. Morgan Kaufmann.

[19] Dyreson, C.E. (2001). Observing transaction-time semantics with *TT*XPath. In WISE.

[20] Faisal, S., Sarwar, M. (2014). Temporal and multi-versioned XML documents: A survey. In Information Processing and Management 50(1): 113-131.

[21] Franconi, E., Meyer, T., Varzinczak. I. (2010). Semantic diff as the basis for knowledge base versioning. In NMR.

[22] Galani, T., Papastefanatos, G., Stavrakas, Y. (2016). A language for defining and detecting interrelated complex changes on RDF(S) knowledge bases. In ICEIS.

[23] Galani, T., Papastefanatos, G., Stavrakas, Y., Vassiliou, Y. (2021). Defining and detecting complex changes on RDF(S) knowledge bases. In Journal on Data Semantics. https://doi.org/10.1007/s13740-021-00136-9

[24] Galani, T., Stavrakas, Y., Papastefanatos, G., Flouris, G. (2015). Supporting complex changes in RDF(S) knowledge bases. In MEPDaW-15 (with ESWC).

[25] Galani, T., Stavrakas, Y., Papastefanatos, G., Vassiliou, Y. (2021). Evo-Path: Querying data evolution through complex changes. In DATA.

[26] Gao, D., Snodgrass, R. T. (2003). Temporal slicing in the evaluation of XML queries. In VLDB.

[27] Gergatsoulis, M., Stavrakas, Y. (2003). Representing changes in XML documents using dimensions. In International XML Database Symposium.

[28] Gonzalez, L., Hogan, A. (2018). Modeling dynamics in semantic web knowledge graphs with formal concept analysis. In WWW.

[29] Grandi, F. (2004). Introducing an annotated bibliography on temporal and evolution aspects in the World Wide Web. In SIGMOD Records.

[30] Guo, Y., Pan, Z., Heflin, J. (2005). LUBM: A benchmark for OWL knowledge base systems. In Journal of Web Semantics 3(2-3): 158-182.

[31] Harris, S., Seaborne, A. (2013, March 21). SPARQL 1.1 Query Language. W3C Recommendation. https://www.w3.org/TR/sparql11-query/. (accessed 3 October 2021)

[32] Kaminski, M., Kostylev, E. V., Cuenca Grau, B. (2017). Query nesting, assignment, and aggregation in SPARQL 1.1. In ACM Transactions on Database Systems 42(3): 17:1-17:46.

[33] Klein, M. (2004). Change management for distributed ontologies. Ph.D. thesis, Vrije University.

[34] Klyne, G., Carroll, J. J., McBride, B. (2014, February 25). RDF 1.1 Concepts and Abstract Syntax. W3C Recommendation. https://www.w3.org/TR/rdf11-concepts/. (accessed 3 October 2021)

[35] Leonardi, E., Bhowmick, S. S. (2005). Detecting changes on unordered XML documents using relational databases: A schema conscious approach. In CIKM.

[36] Leonardi, E., Bhowmick, S. S., Madria, S. (2005). Xandy: Detecting changes on large unordered XML documents using relational databases. In DASFAA.

[37] Maillot, P., Bobed, C. (2018). Measuring structural similarity between RDF graphs. In SIGAPP.

[38] Marian, A., Abiteboul, S., Cobena, G., Mignet, L. (2001). Change-centric management of versions in an XML warehouse. In VLDB.

[39] Meimaris, M. (2016). EvoGen: a generator for synthetic versioned RDF. In EDBT/ICDT Workshops.

[40] Meimaris, M., Papastefanatos, G. (2016). The EvoGen benchmark suite for evolving RDF data. In MEPDaW/LDQ in ESWC.

[41] Moon, H.J., Curino, C., Deutsch, A., Hou, C.Y., Zaniolo, C. (2008). Managing and querying transaction-time databases under schema evolution. In VLDB.

[42] National research council - Committee on Frontiers at the Interface of Computing and Biology (2005). Catalyzing inquiry at the interface of computing and biology. Edited by J. C. Wooley, H. S. Lin., National Academies Press.

[43] Noy, N.F., Musen, M. (2002). PromptDiff: A fixed-point algorithm for comparing ontology versions. In AAAI.

[44] Papastefanatos, G., Stavrakas, Y., Galani, T. (2013). Capturing the history and change structure of evolving data. In DBKDA.

[45] Papavasileiou, V., Flouris, G., Fundulaki, I., Kotzinos, D., Christophides, V. (2013). High-level change detection in RDF(S) KBs. In ACM Transactions on Database Systems 38(1): 1:1-1:42.

[46] Perez, J., Arenas, M., Gutierrez, C. (2009). Semantics and complexity of SPARQL. In ACM Transactions on Database Systems 34(3): 16:1-16:45.

[47] Plessers, P., De Troyer, O., Casteleyn, S. (2007). Understanding ontology evolution: A change detection approach. In Journal of Web Semantics 5(1): 39-49.

[48] Regino, A. G., dos Reis, J. C., Bonacin, R., Morshed, A., Sellis, T. (2021). Link maintenance for integrity in linked open data evolution: Literature survey and open challenges. In Semantic Web 12(3): 517-541.

[49] Rizzolo, F., Vaisman, A. A. (2008). Temporal XML: modeling, indexing, and query processing. In VLDB Journal 17(5): 1179-1212.

[50] Robie, J., Chamberlin, D., Dyck, M., Florescu, D., Melton, J., Simeon, J. (2011, March 17). XQuery Update Facility 1.0. W3C Recommendation. https://www.w3.org/TR/xquery-update-10/. (accessed 3 October 2021).

[51] Robie, J., Dyck, M., Spiegel, J. (2017, March 21). XML Path Language (XPath) 3.1. W3C Recommendation. https://www.w3.org/TR/2017/REC-xpath-31-20170321/. (accessed 3 October 2021)

[52] Robie, J., Dyck, M., Spiegel, J. (2017, March 21). XQuery 3.1: An XML Query Language. W3C Recommendation. https://www.w3.org/TR/2017/REC-xpath-31-20170321/. (accessed 3 October 2021)

[53] Roussakis, Y., Chrysakis, I., Stefanidis, K., Flouris, G., Stavrakas, Y. (2015). A flexible framework for understanding the dynamics of evolving RDF datasets. In ISWC.

[54] Singh, A., Brennan, R., O'Sullivan, D. (2018). DELTA-LD: A change detection approach for linked datasets. In MEPDAW in ESWC.

[55] Stavrakas, Y., Papastefanatos, G. (2010). Supporting complex changes in evolving interrelated web databanks. In CoopIS.

[56] Stavrakas, Y., Papastefanatos, G. (2011). Using structured changes for elucidating data evolution. In DaLi (with ICDE).

[57] Stojanovic, L. (2004). Methods and tools for ontology evolution. Ph.D. thesis, University of Karlsruhe.

[58] Troullinou, G., Roussakis, G., Kondylakis, H., Stefanidis, K., Flouris, G. (2016). Understanding ontology evolution beyond deltas. In MEPDAW in EDBT/ICDT.

[59] Volkel, M., Winkler, W., Sure, Y., Kruk, S., Synak, M. (2005). SemVersion: A versioning system for RDF and ontologies. In ESWC.

[60] Wadler, P. (1999). A formal semantics of patterns in XSLT. In Markup Technologies.

[61] Wang, Y., DeWitt, D. J., Cai, J. (2003). X-Diff: An effective change detection algorithm for XML documents. In ICDE.

[62] Wang, F., Zaniolo, C. (2003). Temporal queries in XML document archives and web warehouses. In TIME.

[63] Wang, F., Zaniolo, C. (2008). Temporal queries and version management for XML document archives. In Data and Knowledge Engineering 65(2): 304-324.

[64] Zablith, F., Antoniou, G., d'Aquin, M., Flouris, G., Kondylakis, H., Motta, E., Plexousakis, D., Sabou, M. (2004). Ontology evolution: A process centric survey. In The Knowledge Engineering Review 30(1): 45-75.

[65] Zeginis, D., Tzitzikas, Y., Christophides, V. (2011). On computing deltas of RDF/S knowledge bases. In ACM Transactions on the Web 5(3): 14:1-14:36.

Γλωσσάρι

causality query	ερώτημα αιτιότητας
change instance	στιγμιότυπο αλλαγής
change modeling	μοντελοποίηση αλλαγών
change definition	ορισμός αλλαγών
change detection	εντοπισμός αλλαγών
coarse-grained change	συνοπτική αλλαγή
complex change	σύνθετη αλλαγή
data/application-agnostic change	αλλαγή που αγνοεί την εκάστοτε εφαρμογή και δεδομένα
data/application-specific change	αλλαγή που είναι συγκεκριμένη για την εκάστοτε εφαρμογή και δεδομένα
data evolution	εξέλιξη δεδομένων
data / dataset version	έκδοση δεδομένων / συνόλου δεδομένων
diff	διαφορά
evolution query	ερώτημα εξέλιξης
fine-grained change	λεπτομερής αλλαγή
first class citizen	πρώτης τάξης πολίτης
framework	πλαίσιο
granularity	βαθμός λεπτομέρειας
human-readable change	αλλαγή κατανοητή από τον άνθρωπο
knowledge base	βάση γνώσεων
machine-readable change	μηχανιστική αλλαγή
pattern	μοτίβο
query	επερώτηση / ερώτημα
querying changes	επερώτηση αλλαγών
reduction process	διαδικασία παραγωγής στιγμιότυπου
semistructured data	ημιδομημένα δεδομένα
simple change	απλή αλλαγή
snapshot	στιγμιότυπο
temporal query	χρονικό ερώτημα

Annex A: Simple Changes in RDF(S) Knowledge Bases

Add_Type_Class(a)	Add object a of type rdfs:Class.
Delete_Type_Class(a)	Delete object a of type rdfs:Class.
Rename_Class(a)	Rename class a to b.
Merge_Classes(A, b)	Merge classes contained in A into b.
Merge_Classes_Into_Existing(A,b)	Merge classes in A into b, $b \in A$.
Split_Class(a,B)	Split class a into classes contained in B.
Split_Class_Into_Existing(a,B)	Split class a into classes in B, $a \in B$.
Add_Type_Property(a)	Add object a of type rdf:property.
Delete_Type_Property(a)	Delete object a of type rdf:property.
Rename_Property(a,b)	Rename property a to b.
Merge_Properties(A,b)	Merge properties contained in A into b.
Merge_Properties_Into_Existing(A, b)	Merge A into b, $b \in A$.
Split_Property(a,B)	Split property a into properties contained in B.
Split_Property_Into_Existing(a,B)	Split a into properties in B, $a \in B$.
Add_Type_Individual(a)	Add object a of type rdfs:Resource.
Delete_Type_Individual(a)	Delete object a of type rdfs:Resource.
Merge_Individuals(A,b)	Merge individuals contained in A into b.
Merge_Individuals_Into_Existing(A,b)	Merge A into b, $b \in A$.
Split_Individual(a,B)	Split individual a into individuals in B.
Split_Individual_Into_Existing(a,B)	Split a into individuals in B, $a \in B$.
Add_Superclass(a,b)	Parent b of class a is added.
Delete_Superclass(a,b)	Parent b of class a is deleted.
Add_Superproperty(a,b)	Parent b of property a is added.
Delete_Superproperty(a,b)	Parent b of property a is deleted.
Add_Type_To_Individual(a,b)	Type b of individual a is added.
Delete_Type_From_Individual(a,b)	Type b of individual a is deleted.
Add_Property_Instance (a1,a2,b)	Add property instance of property b.
Delete_Property_Instance(a1,a2,b)	Delete instance of property b.
Add_Domain(a,b)	Domain b of property a is added.
Delete_Domain(a,b)	Domain b of property a is deleted.
Add_Range(a,b)	Range b of property a is added.
Delete_Range(a,b)	Range b of property a is deleted.
Add_Comment(a,b)	Comment b of object a is added.
Delete_Comment(a,b)	Comment b of object a is deleted.
Change_Comment(u,a,b)	Change comment of resource u from a to b.
Add_Label(a,b)	Label b of object a is added.
Delete_Label(a,b)	Label b of object a is deleted.
Change_Label(u,a,b)	Change label of resource u from a to b.

Annex B: Complex Change Definition Examples in RDF(S) Knowledge Bases

Below, we demonstrate some of the complex change definitions for the EvoGen generated data and the DBpedia data, as defined for the experimental evaluation of the complex change definition language for RDF(S) knowledge bases presented in Chapter 3, Section 3.6.2.2.

1. Class instance additions/deletions
    a. EvoGen

```
CREATE COMPLEX CHANGE Add_UnderGrad_Student(id) {
 CHANGE LIST Add_Type_To_Individual(id, type) ;
 FILTER LIST type=ub:UndergraduateStudent ; } ;
```

    b. DBpedia

```
CREATE COMPLEX CHANGE Add_SoccerPlayer(id) {
 CHANGE LIST Add_Type_To_Individual(id, type) ;
 FILTER LIST type=dbo:SoccerPlayer ; } ;


CREATE COMPLEX CHANGE Delete_SoccerPlayer(id) {
 CHANGE LIST Delete_Type_From_Individual(id, type) ;
 FILTER LIST type=dbo:SoccerPlayer ; } ;
```

2. Property instance additions/deletions
    a. EvoGen

```
CREATE COMPLEX CHANGE Add_Name(id, name) {
 CHANGE LIST Add_Property_Instance(id, prop, name) ;
 FILTER LIST prop=ub:name ; } ;
```

    b. DBpedia

```
CREATE COMPLEX CHANGE Add_Team(id, chId) {
 CHANGE LIST Add_Property_Instance(id, prop, chId) ;
 FILTER LIST prop=dbo:team ; } ;


CREATE COMPLEX CHANGE Delete_Team(id, chId) {
 CHANGE LIST Delete_Property_Instance(id, prop, chId) ;
 FILTER LIST prop=dbo:team ; } ;
```

3. Groupings around added/deleted class instance URIs
    a. EvoGen

```
CREATE COMPLEX CHANGE Add_UnderGrad_Student_Profile(id, name,
univ, email, tel, adv) {
 CHANGE  LIST  Add_UnderGrad_Student(id),  Add_Name(id,  name),
Add_Studing_University(id,    univ),    Add_Email(id,    email),
Add_Telephone(id, tel), Add_Advisor(id, adv) ? ; } ;
```

b. DBpedia

```
CREATE COMPLEX CHANGE Add_SoccerPlayer_with_Details(id, ChId1,
ChId2, ChId3, ChId4, ChId5) {
 CHANGE LIST Add_SoccerPlayer(id), Add_CareerStationProp(id,
chId1)+, Add_Team(id, chId2)*, Add_BirthPlace(id, chId3)*,
Add_Position(id, chId4)*, Add_DeathPlace(id, chId5)* ;
 BINDING LIST union(chId1) as ChId1, union(chId2) as ChId2,
union(chId3) as ChId3, union(chId4) as ChId4, union(chId5) as
ChId5 ; } ;


CREATE COMPLEX CHANGE Delete_SoccerPlayer_with_Details(id,
ChId1, ChId2, ChId3, ChId4, ChId5) {
 CHANGE LIST Delete_SoccerPlayer(id),
Delete_CareerStationProp(id, chId1)+, Delete_Team(id, chId2)*,
Delete_BirthPlace(id, chId3)*, Delete_Position(id, chId4)*,
Delete_DeathPlace(id, chId5)* ;
 BINDING LIST union(chId1) as ChId1, union(chId2) as ChId2,
union(chId3) as ChId3, union(chId4) as ChId4, union(chId5) as
ChId5 ; } ;
```

4. Batch additions/deletions

a. EvoGen

```
Add_UnderGrad_Students_withCommon_Advisor(Ids, adv) {
 CHANGE LIST Add_UnderGrad_Student_Profile(id, name, univ,
email, tel, adv) + ;
 BINDING LIST union(id) as Ids ; } ;
```

b. DBpedia

```
CREATE COMPLEX CHANGE
Add_SoccerPlayers_withCommonPositions(Ids, ChId4) {
 CHANGE LIST Add_SoccerPlayer_with_Details(id, ChId1, ChId2,
ChId3, ChId4, ChId5) + ;
 BINDING LIST union(id) as Ids ; } ;


CREATE COMPLEX CHANGE
Delete_SoccerPlayers_withCommonPositions(Ids, ChId4) {
 CHANGE LIST Delete_SoccerPlayer_with_Details(id, ChId1,
ChId2, ChId3, ChId4, ChId5) + ;
 BINDING LIST union(id) as Ids ; } ;
```

5. Specializations

a. EvoGen

```
CREATE COMPLEX CHANGE Add_Lecturer_withWebAndGradCourses(id,
Courses) {
 CHANGE LIST Add_Lecturer_Courses(id, Courses) ;
 FILTER LIST for some w in Courses : (w, rdf:type,
ub:WebCourse) in Vaf, for some gc in Courses : (gc, rdf:type,
ub:GraduateCourse) in Vaf ; } ;
```

b. DBpedia

```
CREATE COMPLEX CHANGE Add_SoccerPlayer_withTeamIceland(id) {
 CHANGE LIST Add_SoccerPlayer_with_Details(id, ChId1, ChId2,
ChId3, ChId4, ChId5) ;
 FILTER LIST for some c in ChId2 :
c=<http://dbpedia.org/resource/Iceland_national_football_team>
; } ;


CREATE COMPLEX CHANGE Delete_SoccerPlayer_withTeamIceland(id)
{
 CHANGE LIST Delete_SoccerPlayer_with_Details(id, ChId1,
ChId2, ChId3, ChId4, ChId5) ;
 FILTER LIST for some c in ChId2 :
c=<http://dbpedia.org/resource/Iceland_national_football_team>
; } ;
```

6. Updates

   b. DBpedia

```
CREATE COMPLEX CHANGE Update_Team(id, chId1, chId2) {
 CHANGE LIST Add_Team(id, chId1), Delete_Team(id, chId2) ; } ;
```

Annex C: Curriculum Vitae

PERSONAL INFORMATION

Theodora Galani
Knowledge and Database Systems Laboratory
School of Electrical and Computer Engineering
National Technical University of Athens
Iroon Polytechniou 9, Politechnioupoli Zographou
157 80 Athens, Hellas
Phone: (+30) 210 772 1402
Email Address: theodora@dblab.ece.ntua.gr

EDUCATION

2010 - 2021	PhD candidate. National Technical University of Athens. School of Electrical and Computer Engineering. Knowledge and Database Systems Laboratory. Thesis: Managing Evolution in Web Data through Complex Changes. Supervisor: Professor Yannis Vassiliou. Average grade of attended courses: 9.83/10 (Excellent)
2005 - 2010	National Technical University of Athens. Diploma, School of Electrical and Computer Engineering. Major: Computer Science. Average grade: 8.02/10 (Very Good, top 14%) Diploma Thesis: Development of a Proportional-Integral-Differential Algorithm for the control of glucose concentration in patients with Type 1 Diabetes Mellitus. Supervisor: Professor K. S. Nikita.Grade:10.
2005	Graduated from 1st High School of Aegina Final grade: 19.6/20 (Excellent)

RESEARCH INTERESTS

data evolution, change management, web data.

PUBLICATIONS

Theodora Galani, George Papastefanatos, Yannis Stavrakas, Yannis Vassiliou (2021). Defining and detecting complex changes on RDF(S) knowledge bases. In Journal on Data Semantics. https://doi.org/10.1007/s13740-021-00136-9

Theodora Galani, George Papastefanatos, Yannis Stavrakas, Yannis Vassiliou (2021). Evo-Path: Querying Data Evolution through Complex Changes. In10th International Conference on Data Science, Technology and Applications.

Theodora Galani, George Papastefanatos, Yannis Stavrakas (2016). A Language for Defining and Detecting Interrelated Complex Changes on RDF(S) Knowledge Bases. In 18th International Conference on Enterprise Information Systems.

Theodora Galani, Yannis Stavrakas, George Papastefanatos, Giorgos Flouris. Supporting Complex Changes in RDF(S) Knowledge Bases (2015). In 1st International Workshop on Managing the Evolution and Preservation of the Data Web.

Marios Meimaris, George Papastefanatos, Christos Pateritsas, Theodora Galani, Yannis Stavrakas (2015). A Framework for Managing Evolving Information Resources on the Data Web. CoRR abs/1504.06451.

Marios Meimaris, George Papastefanatos, Christos Pateritsas, Theodora Galani, Yannis Stavrakas (2014). Towards a Framework for Managing Evolving Information Resources on the Data Web. In 1st International Workshop on Dataset Profiling & Federated Search for Linked Data.

George Papastefanatos, Yannis Stavrakas, and Theodora Galani. Capturing the History and Change Structure of Evolving Data (2013). In 5th International Conference on Advances in Databases, Knowledge, and Data Applications. Best Paper Award.

FOREIGN LANGUAGES

English	Certificate of Proficiency in English (CPE) – University of Cambridge.
French	Diplôme d' Études en Langue Française (DELF) 1er degré (IFA).

WORKING EXPERIENCE

Jun. 2017-today	Software engineer, AI Labs, EXUS LTD Development and Technical Lead in European projects. Technologies: Java, SpringBoot, ReactJs, MySql/Postgres/MongoDB
Oct. 2015-Mar.2016	Research assistant, Information Management Systems Institute (IMSI), Athena RC. EU-FP7 European Project Diachron. Technologies: RDF(S), SPARQL, Virtuoso, Jena, Java.
Sept. 2014-Sept.2015	Research assistant, Information Management Systems Institute (IMSI), Athena RC. Research Project EICOS, Thales Program. Technologies: XML, XPath/XQuery, Oracle Berkeley DB XML, Java.

TEACHING EXPERIENCE

2011 - 2017	Teaching assistant, School of Electrical and Computer Engineering, National Technical University of Athens. Course: Databases. Instructors: Ass. Professor V. Kantere (2017-2018), Professor N. Koziris (2016-2017), Professor Y. Vassiliou (2013-2016), Professors Y. Vassiliou and Timos Sellis (2011-2013).

TECHNICAL SKILLS

- Java/SpringBoot, ReactJs/Javascript, HTML/CSS
- Sql, MySql, Postgres
- Windows, Linux, Ubuntu
- RDF(S), OWL, SPARQL, Jena, Virtuoso
- XML/XPath/XQuery, Oracle Berkeley DB XML
- IntelliJ IDEA, Visual Studio Code, Eclipse, Microsoft Office & Visio, Matlab

HONOURS AND AWARDS

- Scholarships from the Special Account for Research Grants (ELKE NTUA) for graduate studies (July 2011-June 2014).
- Scholarships from the Institute of Communication and Computer Systems (ICCS) for doctoral research (October-December 2010, October-November 2011, January-April 2013,

January-April 2014, January-May 2015, October 2015-April 2016, October 2016-June 2017).

- Member of the Technical Chamber of Greece (TEE).
- Achievement award from Eurobank EFG for being accepted at university (2005).
- Scholarship award for outstanding performance from the "Ath. Gkikas" foundation (Aegina City, 2005).
- Achievement awards for secondary school years 1999-2005, for excellent school performance.

CONFERENCES

July 2021	Speaker in DATA Conference, Online Streaming.
September 2016	Sub-reviewer in MEDI Conference, Almeria, Spain.
April 2016	Speaker in ICEIS Conference, Rome, Italy.
March 2014	Staff member in EDBT/ICDT Conference, Athens, Greece.
March 2014	Participant in EDF Conference, Athens, Greece.
July 2013	Sub-reviewer in DATA Conference, Reykjavik, Iceland.
June 2011	Staff member in ACM SIGMOD/PODS Conference. Athens, Greece.