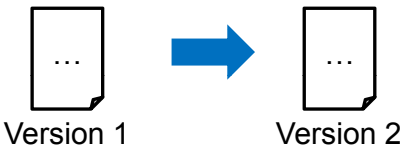
On Detecting High-level Changes in RDF/S KBs

Vicky Papavassiliou, Giorgos Flouris, Irini Fundulaki, Dimitris Kotzinos, Vassilis Christophides

Institute of Computer Science (ICS)
Foundation for Research and Technology Hellas (FORTH), Crete, GREECE



Problem Statement and Motivation



- Visualization of differences
- Evolution history



Efficient storage and/or communication

Why keep track of changes?

- Ontology editors need assistance as manual recording is often neglected, incomplete or erroneous

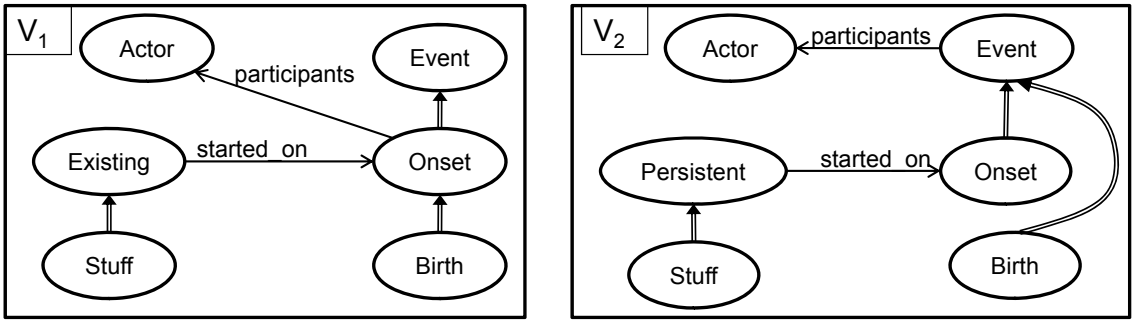
Outline

1. Why high-level changes?
2. Framework for non-heuristic changes
3. Heuristic changes
4. Change Detection Algorithm
5. Evaluation
6. Concluding remarks

Existing approaches

- Deltas based on some language of changes
  - A set of formal definitions that describe the changes that can be understood and detected
  - Can be high-level or low-level
  - Must be coupled with a corresponding detection algorithm
- Low-level languages easy to define (Add(t), Del(t))
- High-level languages more complicated
  - Several proposals; no standard

Why high-level changes?



Carry additional information

Low-level changes

Delete([Existing type class])
Add([Retained type class])
Delete([Stuff subClass Existing])
Add([Stuff subClass Persistent])
Delete([started_on domain Existing])
Add([started_on domain Persistent])

High-level changes

High-level changes

Challenges for High-level Languages

1. Granularity
   - Must be fine-grained enough to capture subtle changes
   - Must be coarse-grained enough to be concise
2. Deterministic Detection Process
   - Must assign to each pair of versions exactly one high-level delta
3. Application Semantics
   - Must be able to apply a high-level delta to obtain a version

Foundation of Proposed Language

- Detection and Definition based on low-level changes
  - Each low-level change is associated to one high-level change
- The **formal definition** of a change consists of:
  1. Required added triples
  2. Required deleted triples
  3. Conditions that should hold in V_1 and/or V_2
- Example: Generalize_Domain(participants,Onset,Event)
  - Add([participants domain Event])
  - Del([participants domain Onset])
  - participants existing property in both V_1, V_2 and Onset, Event existing classes in both V_1, V_2 and Onset subclass of Event in both V_1, V_2

Results on L : Granularity

- Granularity: tackled by defining levels of changes
  - Basic Changes: **fine-grained**, roughly correspond to low-level
  - Composite Changes: **coarse-grained**, group several basic changes together
- Problems
  - One low-level change might be associated with more than one high-level changes
- Solution: Priorities in detection
  - Change domain of property participants from Onset to Event
  - Basic: Delete_Domain(participants,Onset) and Add_Domain(participants,Event)
  - Composite: Generalize_Domain(participants,Onset,Event)

Composite → Basic

Interesting side-effect:

Obtain minimal High-level Delta

Results on L: Determinism

- Completeness
  - Each low-level change is associated with **at least one** detectable high-level change
  - Process any input
- Non-ambiguity
  - Each low-level change is associated with **at most one** detectable high-level change
  - The detected changes are irrelevant to each other

Deterministic Detection Process

Each pair of versions (V_1, V_2) is associated with
- exactly **one** low level delta and
- exactly **one** high-level delta

Results on L: Application

- Application semantics consistent with detection semantics
  - If C = {detected changes between V_1 , V_2 } then $V_1 \bullet C = V_2$
- Independent of order
  - A detected set of changes can be applied in any order and get the same result
  - If C={$c_1 \ldots c_n$} and C' = any permutation over C, then $V_1 \bullet C' = V_2$
- Reversibility
  - Every change in L has a unique reverse
  - Example: Delete_Domain(p,x) = Add_Domain(p,x)$^{-1}$
  - If C = {$c_1 \ldots c_n$} then $V_1 \bullet C = V_2$ and $V_2 \bullet C^{-1} = V_1$

Heuristic changes

- Different because they require matchers
  - Rename_Class(Existing, Persistent)
- Extension to our basic framework and optional
- Detected prior high-level changes
  - No Change_Domain(started_on, Existing, Persistent)
- URIs (Concepts, Properties, …)
  - Neighborhood matcher:
    - 1 to 1 **Rename**
    - 1 to many → **Split**
    - Many to 1 → **Merge**
- Comments and Labels
  - Levenshtein string comparison

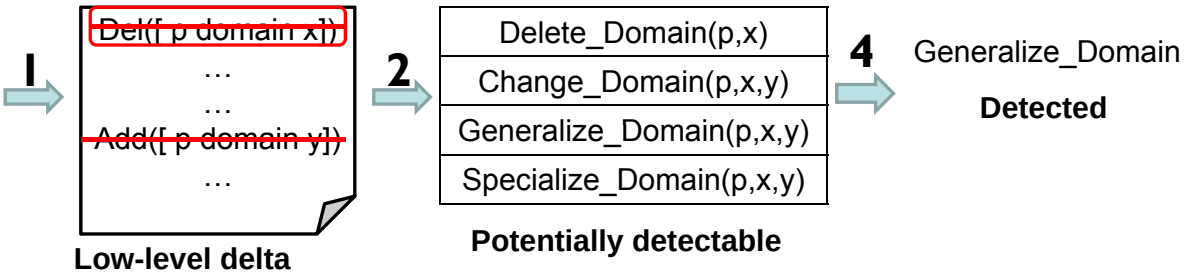
Algorithm: Look-up table

- Principle
  - Every low-level change can potentially trigger the detection of a finite and predetermined subset of high-level changes
- Build look-up table

Low-level change considered	Low-level change(s) searched for	Potential High-level change
Del([p domain x])	-	Delete_Domain(p,x)
Del([p domain x])	Add([p domain y])	Change_Domain(p,x,y)
Del([p domain x])	Add([p domain y])	Generalize_Domain(p,x,y)
Del([p domain x])		Delete_Superclass

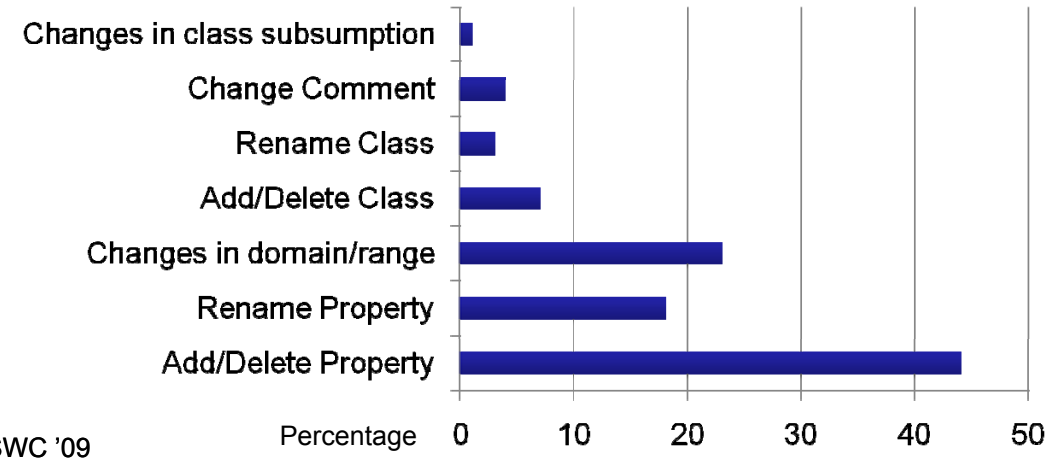
Algorithm: In Practice

1. Select a low-level change from low-level delta
2. Look-up the potentially detectable high-level changes
3. Determine which conditions are true
4. Add it to high-level delta and remove its required added/deleted triples



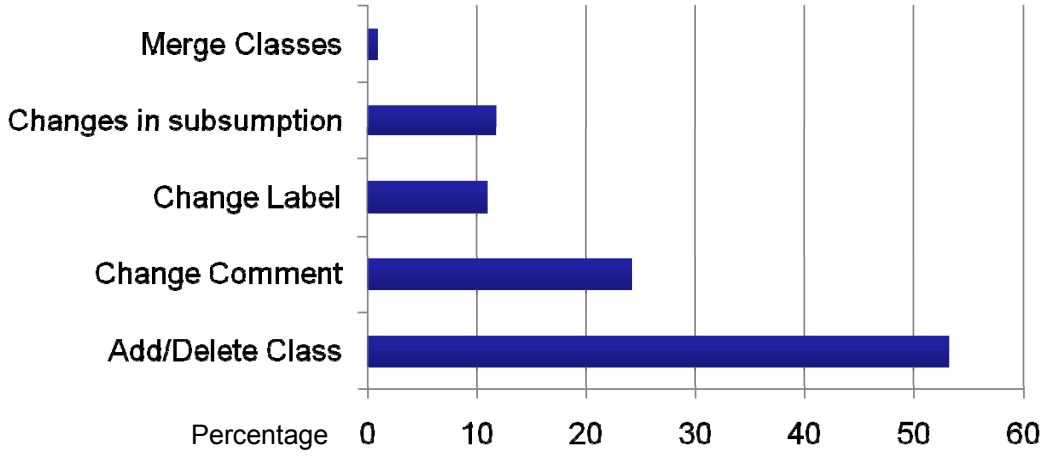
Evaluation: Usefulness

- CIDOC ontology from Cultural domain
  - ~ 80 classes
  - ~ 250 properties
  - ~ 1143 triples
- Results confirmed by CIDOC editor notes



Evaluation: Usefulness

- GO ontology from Biological domain
  - ~ 28 000 classes
  - ~ 1350 property instances all of property *obsolete*
  - ~ 187 587 triples



Evaluation: Conciseness

- Basic ≈ Low-Level
- Basic+Composite+Heuristic << Low-Level

Versions	Low-level Delta	Basic	High-level Delta Basic + Composite + Heuristic
1st pair of Cidoc	870	834	202 + 120 + 39 = 361
2nd pair of Cidoc	287	285	13 + 15 + 34 = 62
3rd pair of Cidoc	571	538	287 + 6 +10 = 303
4th pair of Cidoc	339	327	44 + 51 + 52 = 147
1st pair of GO	2979	2260	326 + 296 + 307 = 929
2nd pair of GO	7312	5053	745 + 706 + 440 = 1891
3rd pair of GO	3108	2322	350 + 362 + 97 = 818
4th pair of GO	2663	1983	265 + 312 + 147 = 724

Evaluation: Algorithm Performance

- Complexity : $O(\max\{N_1, N_2, N_\Delta^2\})$
  - Small overhead
  - Linear average-case
  - Highly dependent on detected changes (type, number)

Versions	Low-level Delta	Basic	Composite	Heuristic
1st pair of Cidoc	95.91 ms	19.53 ms	3.95 ms	26.19 ms
2nd pair of Cidoc	91.45 ms	3.94 ms	1.01 ms	5.54 ms
3rd pair of Cidoc	95.75 ms	8.05 ms	0.26 ms	9.68 ms
4th pair of Cidoc	120.58 ms	5.50 ms	2.12 ms	861.77 ms
1st pair of GO	35.214 s	133.79 ms	28.60 ms	45.195 s
2nd pair of GO	36.610 s	249.66 ms	39.65 ms	345.419 s
3rd pair of GO	36.684 s	146.20 ms	23.99 ms	38.006 s
4th pair of GO	36.712 s	131.22 ms	24.45 ms	40.067 s

Versions	Low-level Delta (Size)	Composite
2nd pair of Cidoc	287	1.01 ms
3rd pair of Cidoc	571	0.26 ms

Conclusion

- A posteriori high-level change detection
  - No further information needed (logs, change recording etc)
  - Non-heuristic based
- Formal semantics
  - Every change formally defined
  - Formal results: Determinism (Completeness, Non-ambiguity), Reversibility, Order Independent Application of Changes
- Efficient, sound and complete detection algorithm
  - No need for precision and recall as heuristics are required by matcher and not by detection
- Interesting informal properties
  - Conciseness, intuitiveness
- Future work: more operations, evaluation on other datasets, evaluation with real users

Thanks for your attention !



Any questions?

Non-Conflicting changes

- Two changes c1,c2 are non-conflicting iff
  - required added triples of c1 *disjoint* required deleted triples of c2
  - required deleted triples of c1 *disjoint* required added triples of c2
- Example
  - Delete_Domain(participants, Event) and Change_Domain(participants, Onset, Event) are conflicting
- The order of application matters!
- A set of high-level changes is conflicting if it contains at least one pair of conflicting changes
- A detected high-level delta is a non-conflicting set of changes



Related Work

- Changes don't have well-defined semantics
  - Not able to reproduce the delta
  - Comparison plausible only on intuitiveness
  - Algorithm cannot be proved to be sound and complete
- Missing framework for the language
  - Completeness and non-ambiguity have not been studied
  - No properties defined
  - Language based on algorithm
- Detection incorporates heuristics



Association of triples to High-level changes

Added triples (Low-level Delta)	Deleted triples (Low-level Delta)	High-level Delta
[participants domain Event]	[participants domain Onset	Generalize_Domain(participants, Onset, Event)
[Birth subClass Event]	[Birth subClass Onset]	Pull_Up_Class(Birth, Onset, Event)
[Persistent type Class]	[Existing type Class]	Rename_Class(Existing, Persistent)
[Stuff subClass Persistent]	[Stuff subClass Existing]	
[started_on domain Existing]	[started_on domain Existing]	



Algorithm Complexity

- Size of low-level delta: N
- Find potentially detectable high-level changes: $O(N^2)$
  1. Size of look-up table constant, time to search: O(1)
  2. Search delta for required triples using hash table: O(N)
  3. Number of returned potentially detectable changes: O(1)
- Evaluate conditions: $O(N^2)$
  1. O(1) <= Number of checks <= $O(M^2)$
     M: required added/deleted triples
  2. Each check: O(1)
- Total complexity: $O(N^2)$



Algorithm: Average Case

- Linear in practice
  1. Search low-level delta using hash table
     - worst case O(N)
     - average case O(1)
  2. Evaluation of conditions
     - worst case $O(M^2)$ depending on type of changes
     - Basic changes: O(1)
  3. Size of M
     - worst case comparable to N