HAL
open science

Algebraic graph transformations for formalizing ontology changes and evolving ontologies

Mariem Mahfoudh, Germain Forestier, Laurent Thiry, Michel Hassenforder

To cite this version:

Mariem Mahfoudh, Germain Forestier, Laurent Thiry, Michel Hassenforder. Algebraic graph transformations for formalizing ontology changes and evolving ontologies. Knowledge-Based Systems, 2015, 73, pp.212-226. 10.1016/j.knosys.2014.10.007 . hal-01162612v2

HAL Id: hal-01162612

https://hal.science/hal-01162612v2

Submitted on 20 Sep 2018

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

This is the author's version of an article published in Knowledge-Based Systems. The final authenticated version is available online at: http://dx.doi.org/10.1016/j.knosys.2014.10.007.

Algebraic graph transformations for formalizing ontology changes and evolving ontologies

Mariem Mahfoudh, Germain Forestier, Laurent Thiry, Michel Hassenforder

MIPS EA 2332, Université de Haute Alsace
12 rue des Frères Lumière 68093 Mulhouse (France)

Abstract

An ontology represents a consensus on the representation of the concepts and axioms of a given domain. This consensus is often reached through an iterative process, each iteration consisting in modifying the current version of the consensus. Furthermore, frequent and continuous changes are also occurring when the represented domain evolves or when new requirements have to be considered. Consequently, ontologies have to be adaptable to handle evolution, revision and refinement. However, this process is highly challenging as it is often difficult to understand all affected ontology parts when changes are performed. Thus, inconsistencies can occur in the ontology as the changes can introduce contradictory axioms. To address this issue, this paper presents a formal approach for evolving ontologies using Typed Graph Grammars. This method relies on the algebraic approach Simple PushOut (SPO) of graph transformations. It formalizes the ontology changes and proposes an a priori approach of inconsistencies resolution. The modified ontology does not need an explicit checking as an incorrect ontology version cannot actually be generated. To validate our proposal, an implementation is presented using the Attributed Graph Grammar (AGG) toolbox.

Keywords:
Ontology Evolution, Typed Graph Grammars, Algebraic Graph Transformations, Consistency, AGG.

1. Introduction

Formalizing knowledge has always presented an existential obsession and an important challenge for humans. The proposed solutions in the literature are mainly organized around databases, data warehouses and more recently ontologies. Ontologies are often defined as *an explicit specification of a conceptualization of a domain* [1]. They make possible for a community to reach a consensus and to bridge the gap of the vocabulary heterogeneity and semantic ambiguities. Thanks to their advantages, ontologies are used in a large range of fields such as: semantic web [2], business decision support [3], image interpretation [4], peer-to-peer networks [5], etc. A counterpart of this popularity, is the constant augmentation of available ontologies. For example, the number of ontologies on the BioPortal increased of 67% in 2013[1]. Furthermore, as building an ontology is an iterative process [6, 7], the creation of a new ontology actually creates a set of several ontologies versions which is also consistently growing. For example, 51 versions of the Gene Ontology (one of the most successfully ontologies) are monthly released since January 2010[2]. Thus, more and more new ontologies are created and the number of versions of existing ontologies is constantly increasing.

Generate a new ontology version is however not a trivial task. It presents several challenges and requires a comprehensive study of the ontology model in order to manage its evolution. *Ontologies Evolution* is defined by Stojanovic et al. as *the timely adaptation of an ontology to the arisen changes and the consistent propagation of these changes to dependent artefacts* [8]. This process consists in the modification of one or many ontology components (class, property, axiom, individual, etc.) and it may be at instances level (*Ontology Population*) and/or structural level (*Ontology Enrichment*) [9]. Moreover, to preserve ontology consistency, the application of ontology changes must preserve all the ontology model constraints [8]. However, ontologies are often developed in a collaborative manner and are usually large and expressive. This makes difficult for a user and/or ontologist to understand all their affected parts (i.e. dependent entities) when changes are made. Therefore, to keep ontology consistency, it is important to have a mechanism that controls how the ontology changes are made and avoids the possible inconsistencies generated due to these changes.

The ontology languages such as Ontology Web Language (OWL[3]) are prevalent in knowledge representation,

[1] bioportal.bioontology.org/ontologies
[2] geneontology.org/ontology-archive
[3] w3.org/TR/owl-ref

although, they are not sufficient for formalizing changes. They are indeed effective to capture static semantics but not changes that require a consistency checking of the interaction between ontologies entities. That is why, the proposed approaches in the literature do not addressed the inconsistencies issue [10] or used an a posteriori process to identify inconsistencies [11, 12, 13], etc. Thus, unlike previous approaches, this paper focuses on the critical issue of presenting a formal approach for consistent ontologies evolution by using Typed Graph Grammars and Algebraic Graph Transformations. Typed Graph Grammars (TGG) are a mathematical formalism that permits to represent and manage graphs. They are used in several fields of computer science such as software systems modelling, pattern recognition and formal language theory [14]. Recently, they started to be used in the ontology field, in particular for the modular ontologies formalization [15], Resource Description Framework graphs representation [16], collaborative ontologies evolution [17] and consistent ontologies evolution [18].

In our previous work [18], we have introduced the formalization of the ontology changes with Typed Graph Grammars and have focused on the atomic changes. A deeper study is presented in this paper which presents an exhaustive list of the atomic ontology changes and describes how consistently formalize the composite and complex changes. A comparison between the ontology changes representation in the OWL and our TGG formalism is presented to highlight the advantages of the use of graph grammars in the ontologies evolution process. Indeed, TGG and algebraic graph transformations provide a new way to formalize ontology changes and offer mechanisms to control graph transformations while avoiding the inconsistencies. Furthermore, they can reduce the number of elementary changes required to apply the composite and complex changes. The proposed approach has been implemented using a graph transformation tool Attributed Graph Grammar (AGG). In addition, we also present a mechanism to log the ontologies versions and ontology changes with a formal representation. An application is presented with the EventCCAlps ontology developed in the frame of the CCAlps European project[4].

The rest of the paper is organized as follow: Section 2 presents related work and introduces Typed Graph Grammars and algebraic graph transformations. Section 3 proposes a graph transformation model for evolving ontologies and describes the formalization of ontology changes with Typed Graph Grammars. Section 4 presents an application using the EventCCAlps ontology. Section 5 evaluates and discusses the proposed approach. Finally, a conclusion summarizes the presented work and gives some perspectives.

[4] ccalps.eu, project reference number: 15-3-1-IT

2. Background and review

2.1. Related work

Managing ontologies evolution has been an important and active field of research in the recent years [9]. The approach of Stojanovic et al. [19] is considered as one of the first works that have addressed this issue. It presents a methodology in six phases: change capturing, change representation, semantics of change, change implementation, change propagation and change validation. The approach focuses on the KAON ontologies and identifies three types of ontology changes: 1) *atomic change* is an ontology change that affects a single ontology entity; 2) *composite change* is an ontology change that modifies the neighbourhood of an ontology entity; 3) *complex change* is an ontology change that can be decomposed into elementary and composite ontology changes. Later, Klein et al. [11] have proposed another classification. They distinguish two types of ontology changes: elementary (atomic) and composite (complex). These changes can be specified via logging of incremental changes or by ontology versions comparison. The authors have also studied the problem of inconsistencies ontologies and proposed strategies resolution for each ontology changes. However, it is important to note that, the work is focused on the "ontology enrichment" and do not specify specific operations for the instances. Then, Luong et al. [20] have addressed both the "ontology enrichment" and the "ontology population". They have studied the evolution management for a corporate semantic web while addressing the RDF[5] (Resource Description Framework) ontologies. This choice restricts the expressivity of the methodology as the others ontology languages (such as OWL) require further types of changes (cardinality changes, restrictions on the classes, etc.). Thus, Djedidi et al. [12] have proposed an approach of OWL ontologies evolution based on pattern conception. They have studied both the atomic and composite changes and have used the Pellet reasoner [21] to detect the inconsistencies. A deeper study of the composite changes is introduced by Javed et al. [22]. It has presented resolution strategies for several composite changes and has described a layered change log for the explicit operational representation of ontology changes. The change log is formalized using a graph-based approach and implemented by OWLAPI[6]. To identify ontologies inconsistencies, Gueffaz et al. [23] have proposed CLOCk (Change Log Ontology Checker) approach which use model checking. A transformation of the OWL ontologies into a specific language NuSMV[7] is needed. However, no strategies are proposed to solve the inconsistencies. Recently, some researches are interested to look for new formalisms to represent ontologies and find others alternatives to the standard ontology languages. Then, Liu et al. [24] have introduced SetPi

[5] w3.org/RDF
[6] owlapi.sourceforge.net
[7] nusmv.fbk.eu

calculus [25] to model ontologies evolution process. They have represented ontologies by using SetPi entities and have defined a new formalism for describing the ontology changes. The work presents many ontology changes (basic and composite). However, it does not study the inconsistencies problem and do not proposes any implementation.

As a summary, various approaches have been proposed to define and implement ontology evolution process. The Table 1 presents a comparison of some approaches according to the languages used, the implementation, the inconsistency management and the specificities. Thus, we can see that different ontology languages have been studied: KAON [8], RDF[20], OWL [11, 12, 22], etc. Based on these languages, several ontology changes were defined and different classification of theses changes were proposed [8, 11]. Despite its importance, the problem of inconsistencies resolution is not sufficiently studied. Indeed, some works do not address this issue [10, 24]. Others approaches are only focused on the inconsistencies identification [23]. Some researches are interested, in addition, to resolve the inconsistencies [12, 20, 22]. However, they use a posteriori process of inconsistencies resolution which require the implementation of changes and then, use external resources to check if the ontology consistency is affected or not. In our work, we propose an a priori approach to avoid inconsistencies by using Typed Graph Grammars formalism.

2.2. Typed Graph Grammars

This section reviews the fundamental notions involved in typed graph grammars and algebraic graph transformations.

Definition 1 (Graph). A graph $G(V, E)$ is a structure composed by a set of vertices V, a set of edges E and an application $s : E \rightarrow V \times V$ that attaches a source/target vertex to each edge.

An attributed graph is a graph extended by a set of attributes name att, a set of possible values val and a mapping valuation $v : att \rightarrow val$.

Definition 2 (Graph Morphism). A graph morphism $m(f, g)$ is an application from a graph $G(V, E)$ to a graph $G(V', E')$ that is defined by two applications $f : V \rightarrow V'$ and $g : E \rightarrow E'$. A morphism must preserve the structure what means that if $e = (s, t)$ and $g(e) = e' = (s', t')$ then $s' = f(s)$ and $t' = f(t)$.

Definition 3 (Typing). A typing is a morphism from a graph $G(V, E)$ to a type graph $TG(V_T, E_T)$ where V_T corresponds to the types of the vertices and E_T to the types of edges.

The Figure 1 gives an example of a graph (lower part) and a morphism/typing to a type graph (upper part).

Definition 4 (Typed Graph Grammars). A typed graph grammar is a formalism defined by $TGG = (G, TG, P)$ where:

- G is a start graph also called host graph.

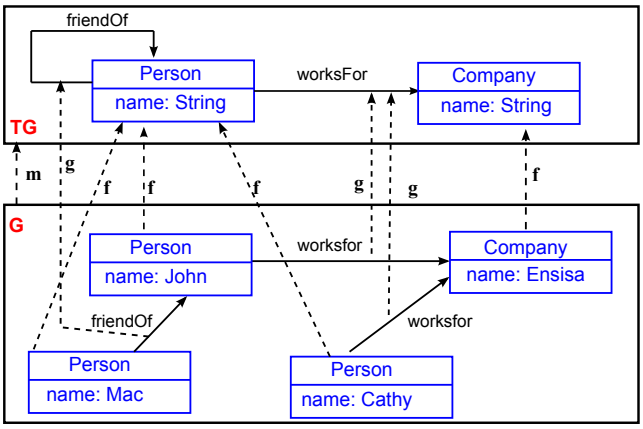


Figure 1: Example of typing graph.

- TG is a type graph and represents the elements type of the graph G.
- P is a set of production rules also called graph rewriting rules (or graph transformations) which are defined by a pair of graphs patterns (LHS, RHS) where:
  - LHS (Left Hand Side) represents the preconditions of the rewriting rule and describes the structure that has to be found in G.
  - RHS (Right Hand Side) represents the postconditions of the rule and must replaces LHS in G.

A rewriting rule can be extended with a set of negative application conditions ($NACs$). A NAC is another graph pattern such as: "if there exist a morphism from NAC to the host graph G, then, the rule cannot be applied". In this way, a graph transformation defines how a graph G can be transformed to a new graph G'. More precisely, there must exist a morphism that replaces LHS by RHS to obtain G'. To apply this replacement, different graph transformations approaches are proposed [26]. In this work, we use the algebraic approach [27] based on the *pushout* concept [28].

Definition 5 (Pushout). The pushout is an operator from the Category Theory [28]. Given three objects (in our case graphs) G_1, G_2 and G_3 and two morphisms $f : G_1 \rightarrow G_2$ and $g : G_1 \rightarrow G_3$, the pushout of G_2 and G_3 consists of: 1) an object G_4 and two morphisms $f' : G_2 \rightarrow G_4$ and $g' : G_3 \rightarrow G_4$ where $f' \circ f = g' \circ g$; 2) for any morphisms $f'' : G_2 \rightarrow X$ and $g'' : G_3 \rightarrow X$ such that $f \circ f'' = g \circ g''$, there is a unique morphism $k : G_4 \rightarrow X$ such that $f' \circ k = f''$ and $g' \circ k = g''$.

Algebraic approaches are divided into two categories: the *Single PushOut, SPO* [29] and the *Double PushOut, DPO* [30]. The DPO approach consists of two pushouts and requires an additional condition called the "dangling

Approach	Ontology Language	Implementation	Inconsistency Management	Specificities
Stojanovic et al., 2004 [8]	KAON Language	KAON framework	- Identification of some inconsistencies. - Strategies proposed to the ontologist to resolve inconsistencies.	- Global evolution process for KAON ontologies. - Saving the evolved version and traceability of the evolution process. - The set of consistency constraints heavily depend on the KAON language.
Klein, 2004 [11]	OWL	OntoView, PROMPTdiff	- Identification and resolve of the inconsistencies.	- Change management approach for distributed ontologies. - Identification of the difference between ontology versions. - Saving the traceability of the ontology changes.
Luong et al., 2007 [20]	RDF(S)	CoSWEM (Corporate Semantic Web Evolution Management)	- Strategies to detect and resolve the inconsistency of the semantic annotations. - Strategies to resolve the ontology changes inconsistencies.	- Evolution management for a corporate semantic web. - Logging management.
Djedidi et al., 2010 [12]	OWL DL	Onto-EVO^{A}L prototype	- Identification inconsistencies using Pellet reasoner.	- Approach based on the pattern conception. - Evaluation of the evolved ontology quality and guiding change resolution. - Approach required heavy activities.
Gueffaz et al., 2012 [23]	OWL DL	Prototype	- Identification inconsistency using NuSMV checker.	- Approach for evolving ontologies based on model checking.
Hartung et al., 2013 [10]	OBO (Open Biomedical Ontologies) and RDF	Conto-diff tool, OnEX web-application	—	- Identification of the difference between ontology versions. - Approach based on the result of a semi-automatic match operation computed by COG (change operation generating) rules.
Khattak et al., 2013 [13]	RDFS and OWL	Protégé plug-in	- Management inconsistencies using KAON API.	- Approach of change history management for evolving ontologies. - Proposition of a Rollback and Rollforward algorithms to revert ontology to the previous or next state respectively based on the logged ontology changes.
Javed et al., 2013 [22]	OWL	OnE (Ontology Editing) tool	- Strategies to detect and resolve the inconsistency of ontology changes.	- A deeper study of the complex ontology changes. - Formalization of the change log using a graph-based approach.
Liu et al., 2014 [24]	OWL	—	—	- Proposition of a new formalism to model ontology evolution, the SetPi Calculus.

Table 1: Summary of ontology evolution approaches.

condition". This condition states that the transformation is applicable only if it does not lead to "dangling edges", i.e. an edge without a source or a target node. Indeed, in the *SPO* approach, one pushout is required and the dangling edges are removed which permits to write a wide variety of transformations not allowed by the *DPO* approach. Thus, in this work, we only consider the *SPO* approach. Applying a rewriting rule to an initial graph (G) with the *SPO* method consists in:

1. Finding a matching of LHS in G, i.e. find a morphism $m : LHS \rightarrow G$.
2. Deleting the sub-graph $m(LHS) - m(LHS \cap RHS)$ from G.
3. Adding the sub-graph $m(RHS) - m(LHS \cap RHS)$ to G to get the final result G'.

An example of rewriting rule is presented in Figure 2: all the persons working in the company "Ensisa" are friends.

3. A graph transformation model for evolving ontologies

In the following, we present our approach for evolving ontologies. We describe the ontology model and the change operations (basic, composite and complex) that may be applied during ontology evolution.

3.1. Ontologies as Typed Attributed Graphs

Due to their mathematical foundation and their application conditions, Typed Graph Grammars are suitable to represent changes and control their effects.

Proposition 1: Ontology representation languages are mainly based on the RDF (Resource Description Framework) model which is based on graphs. Hence, representing ontologies as attributed graphs is quite coherent

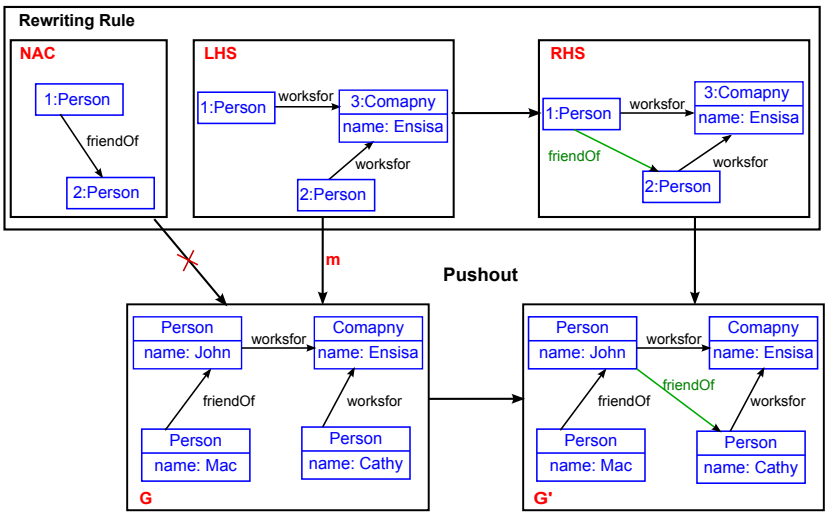


Figure 2: An example of rewriting rule with SPO approach.

and appropriate. In this work we focus on the evolution of OWL ontologies and follow the ontology model axioms, heavily influenced by Description Logics [31]. The OWL was chosen because it is the standard proposed by the W3C and the language usually adopted to represent ontologies. Thus, with the typed graph grammar formalism, an ontology is then, a graph G with a typing relation to type graph (TG) where TG represents the OWL ontology meta-model (Figure 3). Therefore, the considered types of vertices are:

$$V_T = \{Class(C), Property(P), ObjectProperty(OP),\text{-}\\ DataProperty(DP), Individual(I), DataType(D)\}.$$

The edge types correspond to properties used to relate different entities:

$$E_T = \{subClassOf, equivalentTo, range, domain, ...\}.$$

For example, $subClassOf$ is a type of edge that is used to link nodes of the type $Class$. Note that, the restrictions (R) are a special case represented by both nodes ($SomeValuesFrom$, $RestrictionCardinality$, etc.) and edges ($hasRestriction$, $onClass$, etc.).

Both the nodes and the edges can contain attributes. For example, among the attributes of the nodes of types C, I and P, we find the attribute $name$ which specifies their locals names and the iri which identifies them. In the figures of this article, the iri has not represented for readability reasons.

Proposition 2: With proposition 1, ontology changes can be formalized by an indexed family of rewriting rules:

$$r_i = (NAC_i, LHS_i, RHS_i, DCH_i) \text{ where } i \in \\ \{AddClass, RemoveDataProperty, RenameIndividual, ..\}.$$

In this extended definition, DCH represents the set of Derived CHanges to be applied to correct the inconsistencies may be generated due to the application of the ontology changes. For example, the deletion of a class can provoke the modification of its individuals types (i.e. linking these individuals to other classes such as the $superClass$ or the $equivalentClasses$) or the deletion of its individuals as well.

3.2. Formalization of ontology changes

The application of ontology changes can affect ontology consistency. This section describes thus, our proposition for consistent ontologies evolution using the algebraic graph transformations.

Proposition 3: To preserve consistency, each transformation is refined by a set of negative application conditions (NAC) and derived changes (DCH). Theses conditions and additional changes ensure an a priori approach of inconsistencies resolution, i.e. the obtained ontology does not need an explicit checking as an incorrect ontology version cannot actually be generated.

Inspired by the works of literature [32, 33, 34], inconsistencies addressed in this work are:

- Data redundancy that can be generated following an add or rename operation. This type of inconsistency is corrected by the $NACs$.
- Isolated nodes, a node (vertex) V_x called isolated if $\forall V_i \in V, \nexists E_i \in E | E_i = (V_x, V_i)$. This incoherence requires to link the isolated node to the rest of the graph. Depending of the type of node, derived changes are proposed.
- Orphaned individual is an inconsistency which is generated as a result of removal of classes containing individuals.

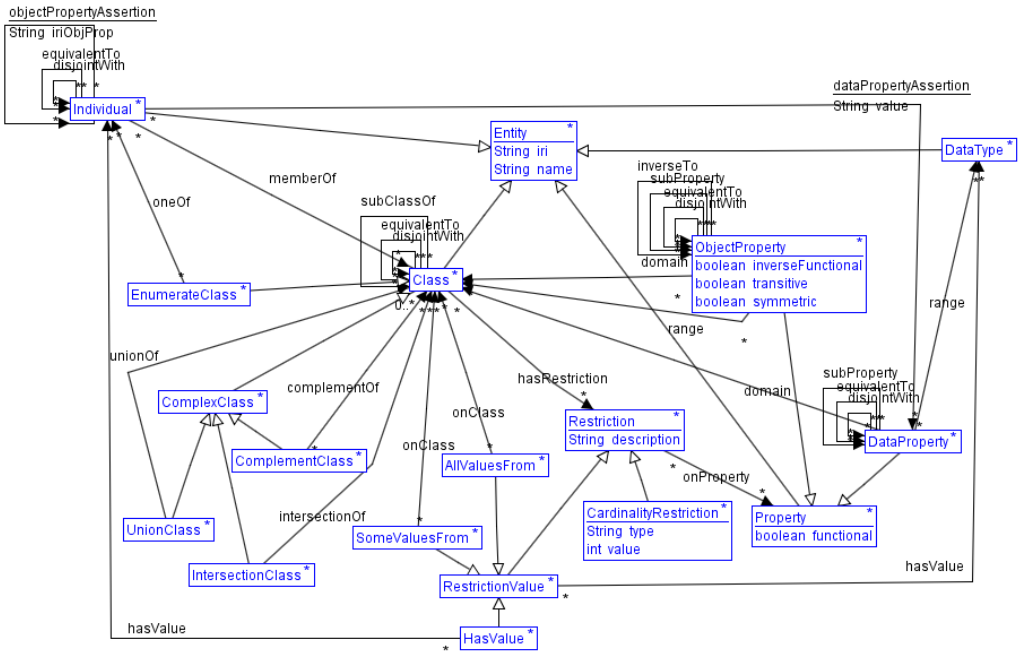


Figure 3: Type Graph used for the rewriting rules formalization.

- Axioms contradiction, the addition of a new axiom should not be accept if it contradicts an axiom already defined in the ontology. Many cases are considered: 1) two classes cannot be disjoint and equivalent at the same time, 2) two classes that share a subsumption relation cannot be disjoints, etc.

3.2.1. Atomic changes

The atomic changes include the rename changes, the addition and deletion of some changes. They only affect a single ontology entity although they depend on other ontologies elements. Thus, the Table 2 presents the atomic changes addressed in this work and the ontology concepts which are related. Actually, it is important to note that the $NACs$ of ontology changes are deduced from these interdependencies. For example, from this table we can see that the $AddDataPropertyAssertion(I,\ DP, value)$ change, which adds a $DataPropertyAssertion$ between an individual I and a dataProperty DP, depends on the $Individual$, $DataProperty$ and $FunctionalProperty$ entities. Indeed, before applying this change, it is necessary to check if the dataProperty DP is a functional property. In this case, if the individual I has already an $AssertionDataProperty$ with the dataProperty DP, then, the change is not allowed because it will affect the ontology consistency.

In the following, one example for each type of change (rename, addition and deletion) is presented. Others changes are described in the Appendix A. Thus, we define for each change, its $NACs$, LHS (pre-condition) and its RHS (post-condition). Of course, this type of change does not have DCH as they affect only a single ontology entity. Note that some changes do not require any NAC such as $RemoveDisjointClasses$, $RemoveEquivalentObjectProperties$, etc.

Thus, the $RenameIndividual(I_i, I_{New})$ is an ontology change that renames a node of type $Individual$. The rewriting rule corresponding to this change is defined as follow (Figure 4a):

- $NAC = \{I_{New}\}$. To avoid redundancy, the NAC of this rule should be the graph composed by a node of type $Individual$ with the attribute $name$ is equal to I_{New}. This means that such sub-graph should not exist in the ontology graph to apply the change.
- $LHS = \{I_i\}$. The LHS represents the pre-condition of a rewriting rule. Thus, in this case, it should be the graph composed by a node of type $Individual$ with the attribute $name$ is equal to I_i. This is necessary to specify that the individual to rename should exist in the ontology.
- $RHS = \{I_{New}\}$. The RHS specifies the new graph that will replace the LHS graph and will be added to the ontology.

The $AddSubClass$ $(C_1,\ C_2)$ rewriting rule adds a $subClassOf$ axiom between two classes (Figure 4c) and it is defined by:

- $NACs$:
  1. $C_1 \sqsubseteq C_2$, condition to avoid redundancy;
  2. $C_2 \sqsubseteq C_1$, the subsumption relation cannot be symmetric;

	Class	Individual	ObjectProperty	DataProperty	DataType	EquivalentClass	DisjointClass	SubClass	EquivalentProperty	DisjointProperty	SubProperty	FunctionalProperty	CardinalityRestriction	AllValuesFromRestriction	SomeValuesFromRestriction	HasValueRestriction
RenameClass	✓															
RenameIndividual		✓														
RenameObjectProperty			✓													
RenameDataProperty				✓												
AddIndividual	✓	✓														
AddDataProperty	✓			✓	✓											
AddObjectProperty	✓		✓													
AddEquivalentClasses	✓	✓				✓	✓									
AddDisjointClasses	✓	✓				✓	✓	✓								
AddSubClass	✓	✓					✓	✓								
AddObjectPropertyAssertion		✓	✓									✓	✓			
AddDataPropertyAssertion		✓		✓								✓				
AddSubObjectProperty			✓							✓	✓					
AddSubDataProperty				✓						✓	✓					
AddCardinalityRestriction	✓		✓										✓			
AddAllValuesFromRestriction	✓		✓											✓		
AddSomeValuesFromRestriction	✓		✓												✓	
AddHasValueRestriction	✓		✓													✓
RemoveIndividual		✓														
RemoveDisjointClasses	✓						✓									
RemoveEquivalentClasses	✓					✓										
RemoveSubClass	✓							✓								
RemoveEquivalentObjectProperties			✓						✓							
RemoveDisjointObjectProperties			✓							✓						
RemoveSubObjectProperty			✓								✓					

Table 2: Matrix dependency between basic ontology changes and ontology entities.

3. $C_1 \sqsubseteq \neg C_2$, classes which share a subsumption relation cannot be disjoint;
4. $\exists C_i \in C(O) \cdot (C_1 \sqsubseteq C_i) \wedge (C_i \sqsubseteq C_2)$, if there is a class C_i which is the $subClassOf$ the class C_2 and the $superClass$ of C_1, then, C_1 is already a $subClass$ of C_2;
5. $\exists (C_i, C_j) \in C(O) \cdot (C_i \sqsubseteq C_1) \wedge (C_j \sqsubseteq C_2) \wedge C_i \sqsubseteq \neg C_j$, classes which share a subsumption relation cannot have subClasses that are disjoint;

- $LHS = \{C_1, C_2\}$, the classes should exist in the ontology.
- $RHS = \{C_1 \sqsubseteq C_2\}$, the axiom will be added to the ontology.

The $RemoveEquivalentObjectProperties(OP_1, OP_2)$ rewriting rule removes the $equivalentTo$ axiom between two objectProperties (Figure 4b) and it is defined by:

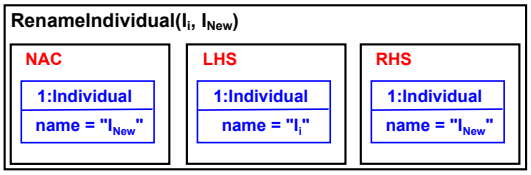
- $NAC = \emptyset$
- $LHS = \{OP_1 \equiv OP_2\}$, the objectProperties and their equivalent relations should exist in the ontology.
- $RHS = \{OP_1, OP_2\}$, the axiom will be removed from the ontology.

3.2.2. Composite changes

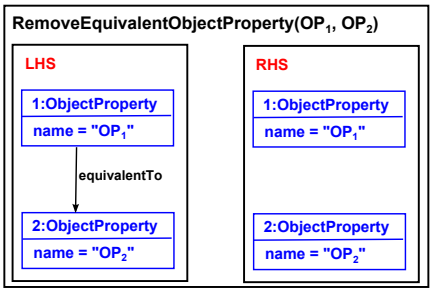
The composite changes affect an ontology entity and its neighbourhood and require then, additional changes ($DCHs$) to preserve the ontology consistency. Thus, the Table 3 shows the interdependencies between theses changes organized as a matrix dependencies. The value of a matrix element (i, j) indicates that the application of a change related to row i involved the application of the changes in column j. In the following, some composite changes are presented.

The $RemoveCardinalityRestriction(C, OP)$ rewriting rule removes a $CardinalityRestriction$ defined on a class C and an objectProperty OP. It is composed of two rules. The first one presents the derived change $RemoveAssertionObjectProperty$ that deletes all the assertions which are defined on OP. The second rule defines the principal rewriting rule that allows the deletion of the restriction.

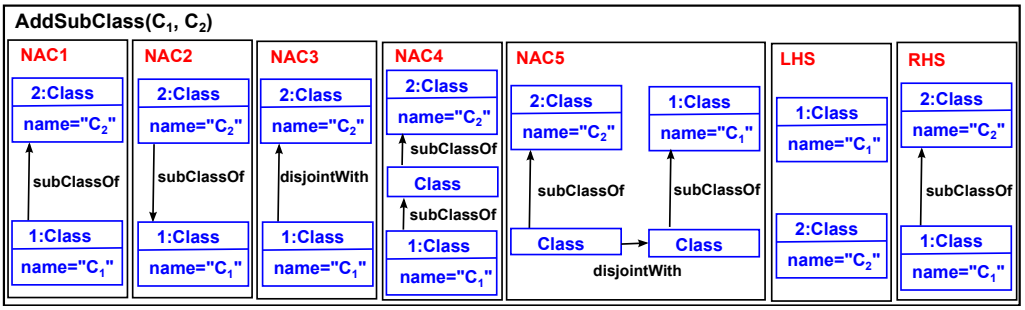
The $RemoveObjectProperty(OP)$ rewriting rule removes an $ObjectProperty(OP)$ and all its dependencies from the ontology. The Figure 5 presents the six rules



(a) Rewriting rule for the *RenameIndividual* change.



(b) Rewriting rule for the *RemoveEquivalentObjectProperty* change.



(c) Rewriting rule for the *AddSubClass* change.

Figure 4: Rewriting rules of some atomic ontology changes.

which define the change. Then, the first five rules describe the derived changes (DCH) must be applied to preserve the consistency of the ontology and the last one presents the principal rewriting rule. Thus, the restrictions defined on the property OP should be all deleted. This is achieved by the application of the following rules: $RemoveAllValuesRestriction(OP)$, $RemoveSomeValuesRestriction(OP)$, $RemoveHasValueRestriction(OP)$ and $RemoveCardinalityRestriction(OP)$. Then, it is necessary to delete all the $ObjectPropertyAssertion$ which reference the objectProperty OP. For the other relations, such as *domain*, *range*, etc., they are directly deleted without needing to add specific controls. The deletion is achieved during the application of the transformation as the SPO approach removes all the dangling edges.

3.2.3. Complex changes

The complex ontology changes are sophisticated operations. They are identified by grouping basic and composite changes and affect several ontology entities which are not necessarily adjacent. They are mostly used to aggregate many and different changes into one in order to perform generic tasks. They help the user to adapt his ontology without being lost in the details of each elementary changes. Table 4 presents the set of complex changes addressed in this work and the changes they are compound of.

The $PullUpClass(C, C_p)$ change moves a class C up in its class hierarchy and attaches it to the parents of its previous parent class C_p. Consequently, the class C is not any more a *subClass* of C_p and thus, does not infer its properties. The figure 6 presents the rewriting rules that defined the change. Thus, the $RemoveObjectPropertyAssertion$ derived change check if the class C has individuals which share an $objectPropertyAssertion$ on the class C_p properties. In this case, all these assertions must be deleted. The $RemoveDataPropertyAssertion$ remove all the $dataPropertyAssertion$ defined on the class C individuals and the class C_p dataProperties.

The $MergeClasses(C_1, C_2, C_{New})$ change merges two exiting classes C_1 and C_2 into a new class (C_{New}). It requires the application of the $AddClass(C_{New})$, $RemoveClass(C_1)$ and $RemoveClass(C_2)$ rewriting rules. However, to preserve the consistency ontology, before deleting C_1 and C_2, all their properties and axioms should be attached to C_{New}. Formally: 1) $\forall C_i \in C(O) \cdot C_i \sqsubseteq C_1$ do the rewriting rule $AddSubClass(C_i, C_{New})$ and $\forall C_j \in C(O) \cdot C_1 \sqsubseteq C_j$ do $AddSubClass(C_{New}, C_j)$, 2) repeat the process with C_2, 3) $\forall C_i \in C(O) \cdot C_i \equiv C_1$ do $AddEquivalentClasses(C_i, C_{New})$, 4) repeat the process with C_2, etc.

The $SplitClass(C, C_{New1}, C_{New2})$ change splits an exiting class (C) into two new created classes C_{New1} and C_{New2}. Then, it requires the application of $AddClass(C_{New1})$, $AddClass(C_{New2})$ and $RemoveClass(C)$ rewriting rules. As the $MergeClasses$ change, the $SplitClass$ rewriting rule requires, before deleting C, the attachment of all its properties and axioms to the C_{New1} and C_{New2}.

	AddClass	RemoveClass	AddSubClass	AddDisjointClasses	AddEquivalentClasses	RemoveIndividual	AddTypeIndividual	RemoveTypeIndividual	RemoveAssertionObjectProperty	RemoveCardinalityRestriction	RemoveAllValuesFromRestriction	RemoveSomeValuesFromRestriction	RemoveHasValueRestriction
AddClass	✓	✓	✓		✓		✓						
RemoveClass		✓				✓	✓	✓	✓	✓	✓	✓	✓
RemoveObjectProperty									✓	✓	✓	✓	
RemoveDataProperty									✓	✓	✓	✓	
RemoveCardinalityRestriction									✓	✓			
RemoveSomeValuesFromRestriction									✓			✓	
RemoveAllValuesFromRestriction									✓		✓		

Table 3: Matrix dependency between composite ontology changes.

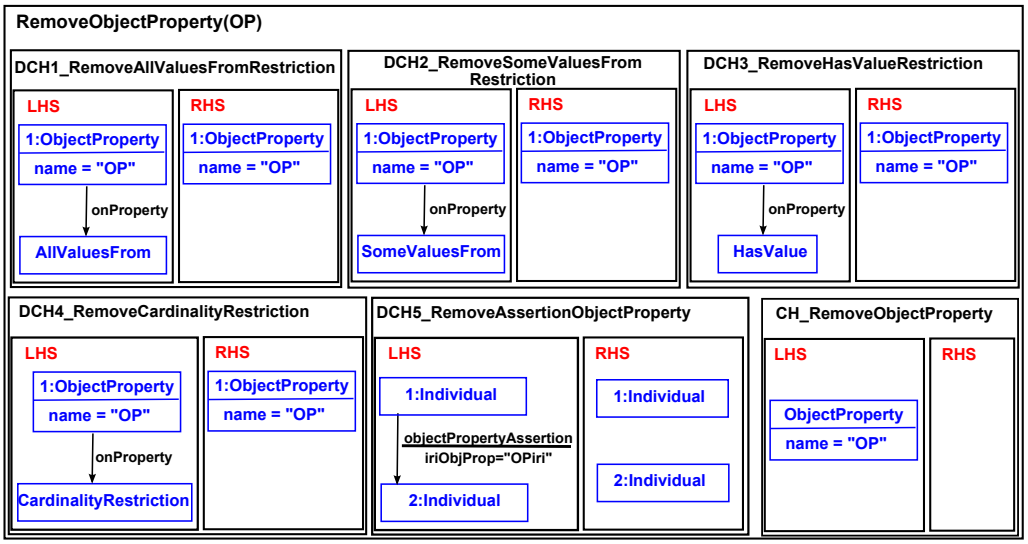


Figure 5: Rewriting rule for the *RemoveObjectProperty* change.

4. Implementation

4.1. EventCCAlps ontology

In this section, we present the EventCCAlps OWL ontology used as a use-case study to describe and validate our research work. The ontology is developed in the frame of the European project CCAlps which aims at helping the collaboration between Creative Companies in Alpine space. It links companies and partners for the organization of events. The Figure 7 presents an extract from the EventCCAlps ontology represented with typed graph attributed formalism. Note that the ontology was converted into AGG graphs using our software OWLToGGX[8]. The Figure shows then, some entities which describe the organized events. An event can be a "Conference", a "Meeting" or a "BestComp". It starts at one day/time and finishes at another day/time, may be described by "Tag" and can receive "Particular" and "Company" participants.

In the EventCCAlps ontology, the changes are frequent both at the instances level (e.g. add events and partners) and schema structure (e.g. create new entities joining the project, delete entities leaving the project).

4.2. Application

Several tools have been proposed to support graph rewriting: AGG [35], Fujaba [36] or Viatra [37]. To implement our approach, we chose the AGG (Attributed Graph Grammar) tool that is considered as one of the most important tools. It supports the algebraic approaches (*SPO*

[8] http://mariem-mahfoudh.info/ksem2013/

		AddClass	RemoveClass	AddSubClass	AddDisjointClasses	AddEquivalentClasses	AddIndividual	AddObjectProperty	RemoveObjectProperty	RemoveObjectPropertyAssertion	RemoveDataPropertyAssertion	AddSubProperty	AddEquivalentProperties	AddDisjointProperties
PullUpClass			✓							✓	✓			
MergeClasses	✓	✓	✓	✓	✓	✓								
SplitClass	✓	✓	✓	✓	✓	✓								
SplitObjectProperty							✓	✓				✓	✓	✓
MergeObjectProperties							✓	✓				✓	✓	✓

Table 4: Matrix dependency of complex ontology changes.

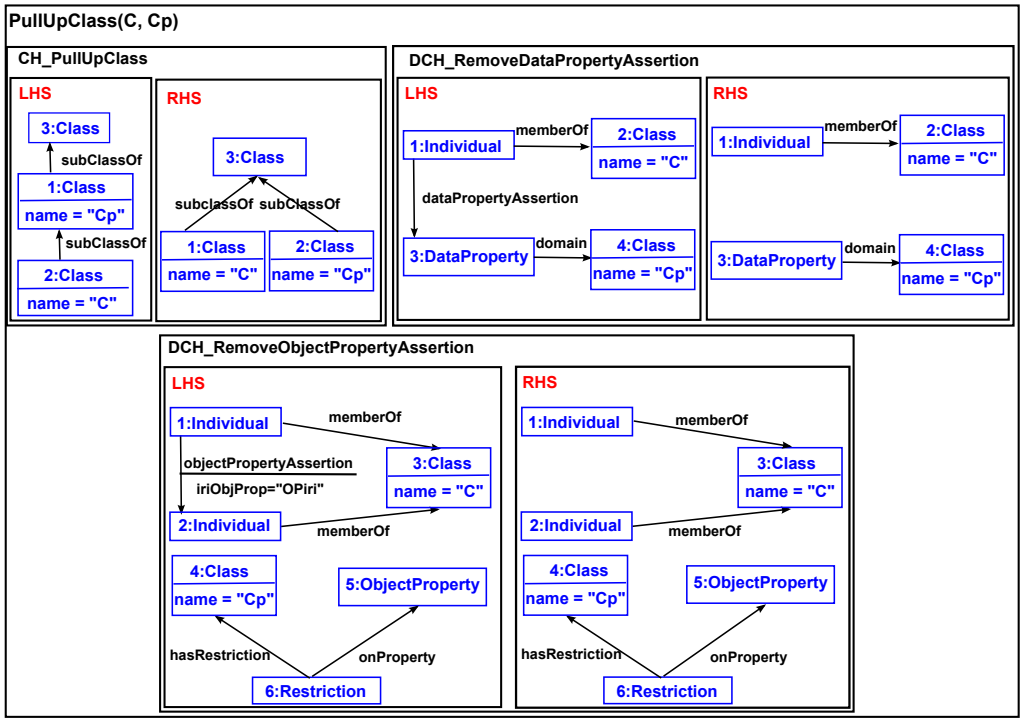


Figure 6: Rewriting rule for the *PullUpClass* change.

and *DPO*) and provides graphical editors for graphs and rewriting rules. The Figure 8 shows the AGG graphical user interface and presents how to implement ontology changes[9]. Different editors are shown: A) an editor for node and edge types that allows user to specify the elements of the type graph; B) a graphical editor for graphs that creates and shows the host graph and type graph; C) a graphical editor for rewriting rules that permits to define the *NACs*, *LHS* and *RHS* of each rule. An example of grammar graph is also presented. It is namely GraphTransformationSystem and it is consisting of: 1) the type graph that presents the ontology meta-model; 2) the host graph that contains the ontology to be evolved; 3) two rewriting rules (*AddDataPropertyAssertion* and *AddDisjointClasses*).

Note that the rewriting rules corresponding to the composite and complex changes are classified by layers to define the sequence of their application.

Now, we present real cases study with the CCAlps project. Thus, as mentioned above, the EventCCAlps ontology defines events whose the participants may be "Particular" or "Company". However, due to the part-

[9]All the materials used in this section (ontology in graph (AGG format) along with the code of the presented ontology changes) are available for download here: http://mariem-mahfoudh.info/kbs2014/

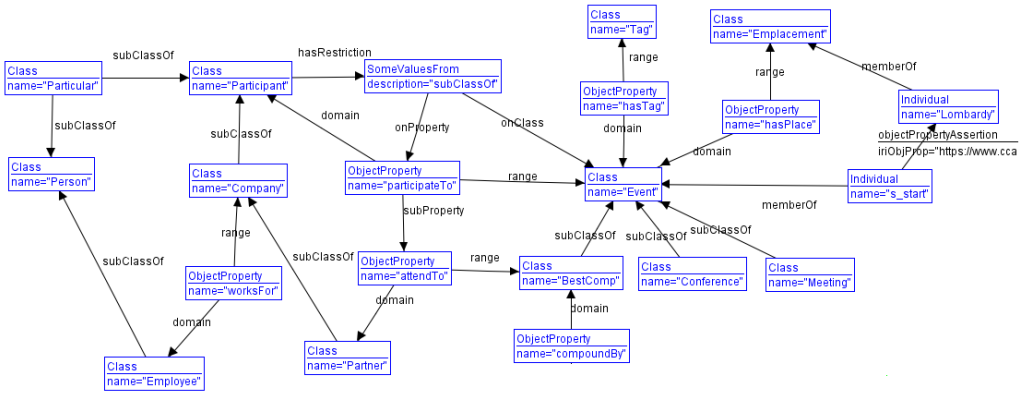


Figure 7: An extract from EventCCAlps ontology represented with the typed attributed graph formalism.

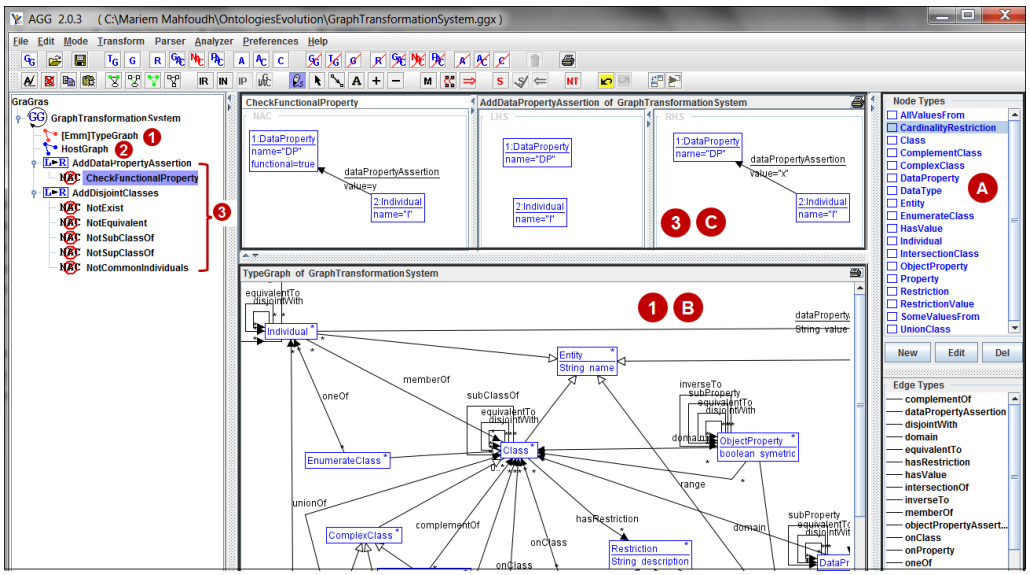


Figure 8: AGG graphical user interface.

ners requirements, it was necessary to distinguish between the companies types and thus replace the "Company" entity by the "CCI" and "NotCCI" concepts where CCI (Creative and Cultural Industries) are the companies whose the activities take origin from individual creativity such as performing arts, graphic design, etc. Thus, the rewriting rules corresponding to this change is $SplitClass(Company, CCI, NotCCI)$. The Figure 9 presents the different rules required to apply the change as described in the Section 3.2.3 and shows the ontology result. Therefore, the class "Company" are deleted and all its properties and axioms are attached to the "CCI" and "NotCCI" classes. These new classes have now, a subclass "Partner", a super-class "Participant" and they are connected by the "worksFor" property.

The Figure 10 presents the $AddDisjointClasses(Meeting, Event)$ change that adds a disjunction axiom between the two classes "Meeting" and "Event". The rewriting rule is composed by five NACs: 1) NAC 1 avoids redundancy; 2) NAC 2 forbids the application of the rule if the classes "Meeting" and "Event" are equivalent, because classes cannot be disjoint and equivalent at the same time; 3) NAC 3 prohibits the transformation if the class "Event" is a subclass of the class "Meeting". Classes that share subsumption relation cannot be disjoint; 4) NAC 4 forbids the transformation if the class "Meeting" is a subclass of the class "Event"; 5) NAC 5 forbids the application of the rule if the classes have common individuals.

As the classes "Meeting" and "Event" share a subsumption relation (a "Meeting" is a subClassOf "Event") then, the rewriting rule cannot be applied (violation of the NAC 4) and an alert box appears to inform user that the transformation cannot be achieved.

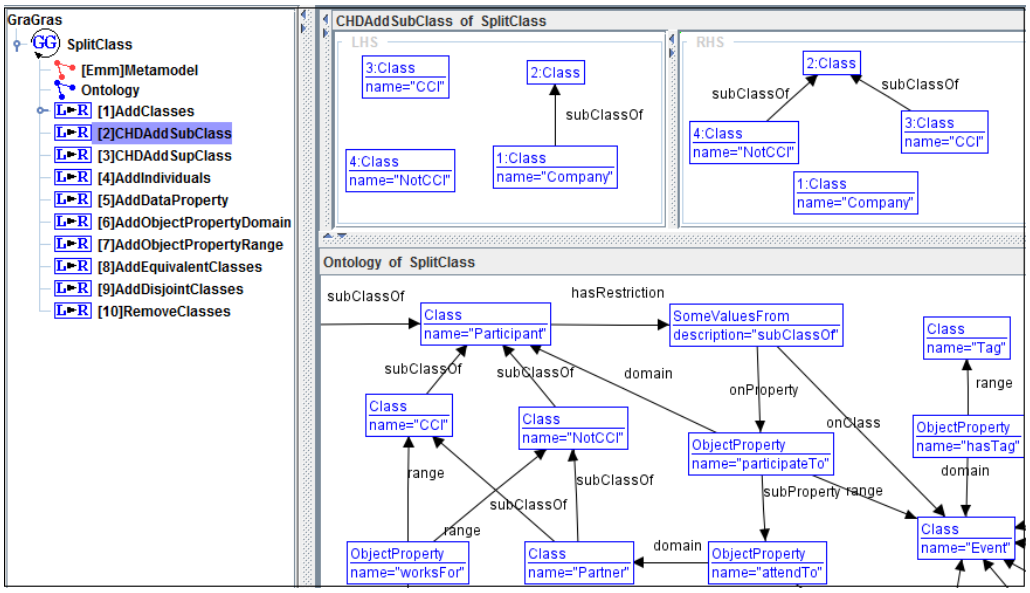


Figure 9: $SplitClass(Company, CCI, NotCCI)$ rewriting rule.

5. Results and discussion

5.1. Formalisms comparison

The two main advantages of our method are: (1) to provide a new way to formalize ontology changes while controlling the graph transformations and avoiding the inconsistencies with an a priori manner; (2) to facilitate the description of composite and complex changes while reducing the number of the rewriting rules required to apply them. In order to highlight these two main features, we present in this section a comparison between the ontology changes representation in the *TGG* formalism proposed in this paper and the closest related approach: Djedidi et al. [12] (see Section 2.1).

The Table 5 presents two examples of ontology changes: $AddObjectProperty$ and $PullDownClass$. In [12], the changes are considered as composite/complex. The first one is composed of three basic changes and the second one of two basic changes. The execution of the ontology changes requires the use of Pellet reasoner which is used as an external resource to identify the inconsistencies that can appear. Therefore, the inconsistencies resolution is achieved by an *a posteriori* manner. If the ontology consistency is affected, the changes must be canceled to go back to the previous ontology version. In our work, these changes are considered as elementary changes as they are composed by only one rewriting rule. Moreover, to preserve the ontology consistency, the checking of the inconsistencies is achieved by the negative application conditions (NAC) which ensure an *a priori* approach of inconsistencies resolution. Thus, there is no need of an external resource to check the consistency of the ontology as the entire the evolution process is supported by the TGG formalism.

5.2. Complexity

The most expensive step in time and resource of the proposed approach is the recognition of the LHS from the host graph G. This search is an NP-complete problem. More precisely, a search of a sub-graph composed of k elements in a graph compound of n elements has a complexity of $O(n^k)$. However, the cost of calculation remains quite acceptable if the size of the LHS graph is limited [38]. This condition is generally satisfied in ontology changes application. The number of nodes of the LHS graph can be used as a measure of the complexity of the ontology change. As presented in Table 6, the LHS size is quite limited for simple change (0 to 3 nodes). For more complex changes, the DHC size has also to be considered.

The execution time is also dependent of the size of the LHS and the ontology's change type (simple or complex). In the example presented in this paper (see section 4.1), the ontology's graph is composed of 21 nodes. The execution of the $AddDisjointClasses(Event, Meeting)$ change took 10 milliseconds (with a LHS composed of 2 nodes). The execution of the complex ontology change $SplitClasses(Company, CCI, NotCCI)$ took only 700 milliseconds (with a LHS composed of 37 nodes). These execution times are quite acceptable as they offer a real-time feedback when executing changes on small-sized ontologies.

5.3. Discussion

Ontology Changes Classification. In our previous work [18], we distinguished the ontology changes considering the classification proposed by Klein et al. [11]. Indeed, this classification of basic/elementary and composite/complex changes is based on the user's vision and does not take into consideration the system's vision. That is why some changes, such as $RemoveClass$, are considered

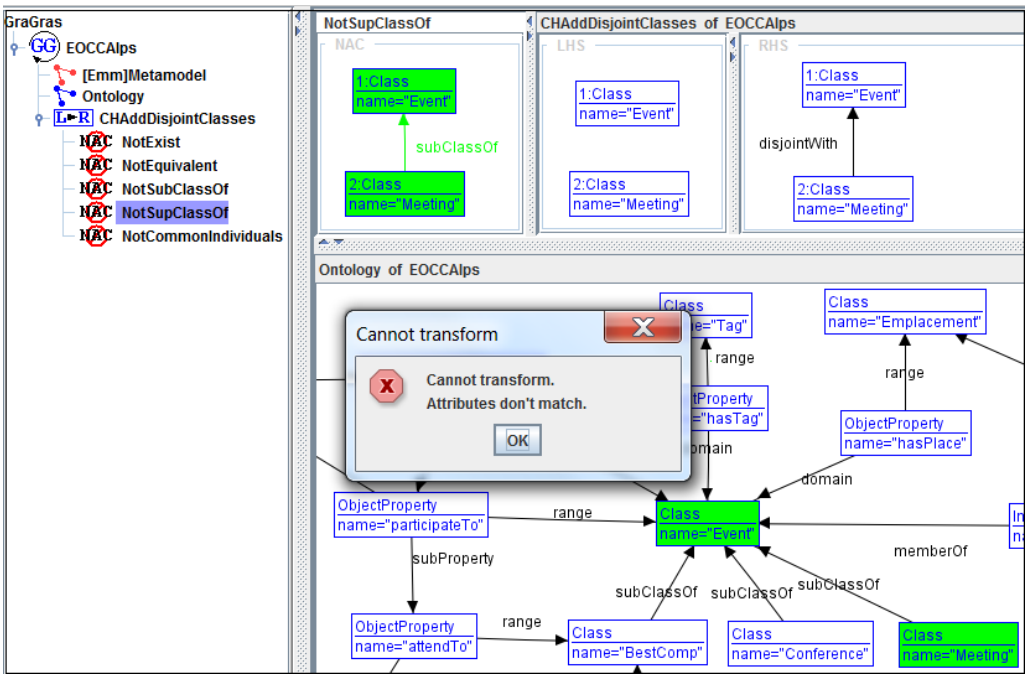


Figure 10: $AddDisjointClasses(Meeting, Event)$ rewriting rule.

as basic changes (user's vision) although at the system level they are rather composite since they call for corrective operations. For example, the *RemoveClass* change involved others basic changes as *RemoveRestriction*, *RemoveIndividual*, etc. Therefore, in this work we have reclassified the ontology changes and make the distinction between basic, composite and complex changes by considering the system's vision.

Logging. Saving and storing the changes for later use is an important task in ontology evolution as this type of information may be useful in the management of the distributed and dependent ontologies. Indeed, by using the AGG tool, we can preserve after each application of ontology change, a log file that stores the type graph, the host graph (i.e. the ontology) and the rewriting rules in a formal and semantic representation (Figure 11). Currently, in our work, all the applied changes as well as the different ontology versions can be recovered. The Figure 12 shows how storing the versions of the same ontology with AGG tool. Thus, at t_0, we have the first ontology version (O_{V0}). Then, when the ontology changes are requested, another file is generated to store both the ontology and the rewriting rules ($O_{V0} + R_{VoToV1}$). After applying ontology changes, a new ontology version will be generated including the rewriting rules ($O_{V1} + R_{VoToV1}$). This makes possible the identification of the difference between two ontologies versions, providing that the changes are defined by our methodology. However, a further study should be carried out to optimize the storing of the various versions and to answer to the following questions: us should we save all the ontology versions? How to identify and choose the most relevant versions?

6. Conclusion

In this paper, we have proposed Typed Graph Grammar and Algebraic Graph Transformations to formalize and manage ontologies evolution. Several ontology changes (basic, composite and complex) were presented and an a priori approach of inconsistencies resolution was introduced. Thanks to the negative application conditions (NAC) and derived changes (DCH), our method avoids the inconsistencies and preserve the evolved ontology quality. The use of algebraic graph transformations offers several advantages. In particular, it allows to simply and formally define the rewriting rules corresponding to the ontology changes. Moreover, it reduces the number of elementary changes required to apply the composite and complex changes providing thus, a gain in time and resources. The execution time of most of the ontology changes are quite limited (< 1 second). To further evaluate the performance of our approach, we are currently working on evaluating the influence of the size of LHS on larger ontologies.

In the near future, we are planning to develop a plug-in for the Protege[10] editor which will present the different ontology changes. Indeed, the internal graph transformation engine of the AGG tool can be used by a Java API and thus, be integrated into other custom applications. Then, we intend to benefit from the formal operations of graph theory and category theory to study, without using

[10] protege.stanford.edu

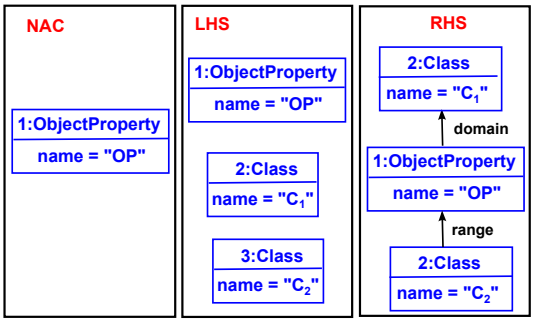
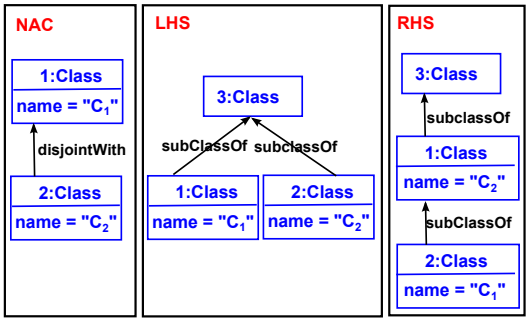
Ontology Changes	Djedidi et al. [12]	Proposed formalism
$AddObjectProperty(OP, C_1, C_2)$	The change is composed of three basic changes: 1. $AddObjectProperty(OP)$, 2. $AddDomain(OP, C_1)$ 3. $AddRange(OP, C_2)$.	- The change is formalized by one rewriting rule. - The rule forbids the data redundancy. 
$PullDownClass(C_1, C_2)$: move down a class (C_1) in its class hierarchy and attach it as a child to its previous sibling class (C_2).	The change is composed of two basic changes: 1. $AddSubClass(C_1, C_2)$ 2. $RemoveSubClass(C_1, C_p)$ where C_p is the parent class of C_1 and C_2.	- The change is formalized by one rewriting rule. - The rule forbids the application of the change if the classes C_1 and C_2 are disjoint as classes that share a subsumption relation cannot be disjoint. 

Table 5: Ontology changes formalization according to Djedidi et al. [12] and the proposed approach.

Ontology changes	Size of LHS
$AddClass$, $AddIndividual$	0 nodes
$RenameClass$, $RenameIndividual$, $RenameDataProperty$	1 nodes
$AddEquivalentClasses$, $AddDisjointClasses$, $AddSubClasses$	2 nodes
$AddObjectProperty$, $AddDataProperty$	3 nodes
$PullUpClass$	At least 3 nodes for the principal change. At least 4 nodes for the first DCH. At least 6 nodes for the second DCH.

Table 6: The LHS's size of some ontology changes.

the logging file, the difference between ontology versions and the automatic detection of the ontology changes. This study may be inspired by both the existent works in the meta-modelling [39] and ontologies [10].

Acknowledgment

The authors would like to thank the European project CCAlps which funded this work (project number is 15-3-1-IT).

Appendix A. Ontology changes formalized by TGG

References

[1] T. R. Gruber, A translation approach to portable ontology specifications, Knowledge acquisition 5 (2) (1993) 199–220.

[2] C. Zhang, C. Cao, Y. Sui, X. Wu, A chinese time ontology for the semantic web, Knowledge-Based Systems 24 (7) (2011) 1057 – 1074.

[3] Y. Zhao, Z. Li, X. Wang, W. A. Halang, Decision support in e-business based on assessing similarities between ontologies, Knowledge-Based Systems 32 (0) (2012) 47 – 55.

Ontology Change	NACs	LHS	RHS	DCHs
$AddDisjointClasses(C_1, C_2)$	1. $C_1 \sqsubseteq \neg C_2$; 2. $C_1 \equiv C_2$; 3. $C_1 \sqsubseteq C_2$; 4. $C_2 \sqsubseteq C_1$; 5. $\exists I_i \in I(O) \cdot I_i \in C_1 \wedge I_i \in C_2$.	$\{C_1, C2\}$	$\{C_1 \sqsubseteq \neg C_2\}$	—
$RemoveDisjointClasses(C_1, C_2)$	—	$\{C_1 \sqsubseteq \neg C_2\}$	$\{C_1, C_2\}$	—
$AddEuivalentClasses(C_1, C_2)$	1. $C_1 \sqsubseteq \neg C_2$; 2. $C_1 \equiv C_2$.	$\{C_1, C_2\}$	$\{C_1 \equiv C_2\}$	—
$RemoveEquivalentClasses(C_1, C_2)$	—	$\{C_1 \equiv C_2\}$	$\{C_1, C_2\}$	—
$AddDomainDataProperty(DP_i, C_i)$	—	$\{DP_i, C_i\}$	$\{DP_i \sqsubseteq C_i\}$	—
$RemoveDomainDataProperty(DP_i, C_i)$	—	$\{DP_i \sqsubseteq C_i\}$	$\{DP_i, C_i\}$	—
$AddRangeDataTypeClass(DP_i, D_i)$	—	$\{DP_i, D_i\}$	$\{\top \sqsubseteq \forall DP_i.D_i\}$	—
$RemoveRangeDataType(DP_i, D_i)$	—	$\{\top \sqsubseteq \forall DP_i.D_i\}$	$\{DP_i, D_i\}$	—
$AddObjectPropertyAssertion(I_1, I_2, OP_i)$	1. $(I_1, I_2) \in OP_i$; 2. $\exists I_i \in I(O) \cdot (I_i \neq I2) \wedge ((I_1, I_i) \in OP_i) \wedge (\top \sqsubseteq 1OP_i)$; 3. $\exists \leqslant nOP_i \cdot (\exists I_i \in I(O)) \wedge \{(I_1, I_i) \in OP_i\} = n$.	$\{I_1, I_2, OP_i\}$	$\{(I_1, I_2) \in OP_i)\}$	—
$RemoveObjectPropertyAssertion(I_1, I_2, OP_i)$	—	$\{(I_1, I_2) \in OP_i)\}$	$\{I_1, I_2, OP_i\}$	—
$AddIndividual(I_{New}, C_i)$	$\{I_{New}\}$	$\{C_i\}$	$\{I_{New} \in C_i\}$	—
$RemoveIndividual(I_i)$	—	$\{I_i\}$		—
$AddDataProperty(DP_{New}, C_i, D_i)$	$\{DP_{New}\}$	$\{C_i, D_i\}$	$\{DP_{New} \sqsubseteq C_i \wedge \top \sqsubseteq \forall DP_i.D_i\}$	—
$RemoveDataProperty(DP_i)$	—	$\{DP_i\}$		—
$AddClass(C_{New})$	$\{C_{New}\}$		$\{C_{New}\}$	1. $AddSubClass$ 2. $AddDisjointClasses$ 3. $AddEquivalentClasses$ 4. $AddDomain$ 5. $AddRange$
$RemoveClass(C_i)$		$\{C_i\}$		1. $SetTypeIndividual$ 2. $RemoveIndividual$ 3. $RemoveRestrictions$

Table A.7: Formalization of ontology changes with TGG.

```
<Document version="1.0">
 − <GraphTransformationSystem ID="I1" directed="true" name="OntologyChanges" parallel="true">
   − <Types>
     − <NodeType ID="I2" name="Entity%[NODE]:">
         <AttrType ID="I4" attrname="iri" typename="String"/>
         <AttrType ID="I5" attrname="name" typename="String"/>
       </NodeType>
     − <NodeType ID="I6" name="Class%[NODE]:">
         <Parent pID="I2"/>
       </NodeType>
       ...
       <EdgeType ID="I49" name="subClassOf%[EDGE]:"/>
       <EdgeType ID="I50" name="disjointWith%[EDGE]:"/>
       ...
     − <Graph ID="I69" kind="TG" name="OntologyMetamodel">
         <Node ID="I72" type="I6"/>
         <Node ID="I74" type="I2"/>
         ...
         <Edge ID="I106" source="I49" target="I6" type="I6"/>
         ...
       </Graph>
   </Types>
 − <Graph ID="I137" kind="HOST" name="Ontology">
   − <Node ID="I138" type="I6">
     − <Attribute constant="true" type="I4">
       − <Value>
           <string>Ciri</string>
         </Value>
       </Attribute>
     − <Attribute constant="true" type="I5">
       − <Value>
           <string>C</string>
         </Value>
       </Attribute>
     </Node>
     ...
   </Graph>
 − <Rule ID="I138" formula="true" name="CHRenameClass">
   − <Graph ID="I140" kind="LHS" name="LeftOf_CHRenameClass">
     − <Node ID="I141" type="I6">
       − <Attribute constant="true" type="I4">
         − <Value>
             <string>Ciri</string>
           </Value>
         </Attribute>
       − <Attribute constant="true" type="I5">
         − <Value>
             <string>C</string>
           </Value>
         </Attribute>
       </Node>
     </Graph>
     <Graph ID="I145" kind="RHS" name="RightOf_CHRenameClass">
     ... </Graph>
   − <Morphism comment="Formula: true" name="CHRenameClass">
       <Mapping image="I146" orig="I141"/>
     </Morphism>
   − <ApplCondition>
     − <NAC>
       − <Graph ID="I150" kind="NAC" name="NotExist">
         − <Node ID="I151" type="I6">
           − <Attribute constant="true" type="I4">
             − <Value>
                 <string>CiriNew</string>
               </Value>
             </Attribute>
             ...
           </Node>
         </Graph>
         <Morphism name="NotExist"/>
       </NAC>
     </ApplCondition>
     <TaggedValue Tag="layer" TagValue="0"/>
     <TaggedValue Tag="priority" TagValue="0"/>
   </Rule>
   <Rule ID="I155" formula="true" name="AddIndividual"> ... </Rule>
```


Figure 11: An extract from the log file of ontology changes.

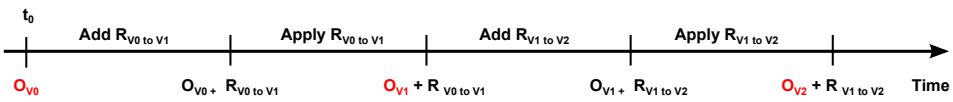


Figure 12: Storage ontology versions with AGG tool.

[4] G. Forestier, C. Wemmert, A. Puissant, Coastal image interpretation using background knowledge and semantics, Computers & Geosciences 54 (2013) 88–96.

[5] J. J. Jung, Reusing ontology mappings for query routing in semantic peer-to-peer environment, Information Sciences 180 (17) (2010) 3248 – 3257.

[6] N. F. Noy, D. L. McGuinness, et al., Ontology development 101: A guide to creating your first ontology (2001).

[7] R. Thomopoulos, S. Destercke, B. Charnomordic, I. Johnson, J. Abécassis, An iterative approach to build relevant ontology-aware data-driven models, Information Sciences 221 (2013) 452–472.

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

[9] A. M. Khattak, R. Batool, Z. Pervez, A. M. Khan, S. Lee, Ontology evolution and challenges, Journal of Information Science and Engineering 29 (2013) 851–871.

[10] M. Hartung, A. Groß, E. Rahm, Conto–diff: generation of complex evolution mappings for life science ontologies, Journal of Biomedical Informatics 46 (1) (2013) 15–32.

[11] M. Klein, Change management for distributed ontologies, Ph.D. thesis, Vrije Universiteit Amsterdam, Amsterdam, The Netherlands (2004).

[12] R. Djedidi, M.-A. Aufaure, *ONTO-EVO*^{al} an ontology evolution approach guided by pattern modeling and quality evaluation, in: Foundations of Information and Knowledge Systems, Springer, 2010, pp. 286–305.

[13] A. M. Khattak, K. Latif, S. Lee, Change management in evolving web ontologies, Knowledge-Based Systems 37 (0) (2013) 1–18.

[14] H. Ehrig, U. Montanari, G. Rozenberg, H. J. Schneider, Graph Transformations in Computer Science, Geschäftsstelle Schloss Dagstuhl, 1996.

[15] M. d'Aquin, P. Doran, E. Motta, V. A. Tamma, Towards a parametric ontology modularization framework based on graph transformation., in: WoMO, 2007.

[16] B. Braatz, C. Brandt, Graph transformations for the resource description framework, Electronic Communications of the EASST 10.

[17] P. De Leenheer, T. Mens, Using graph transformation to sup-

port collaborative ontology evolution, in: Applications of Graph Transformations with Industrial Relevance, Springer, 2008, pp. 44–58.

[18] M. Mahfoudh, G. Forestier, L. Thiry, M. Hassenforder, Consistent ontologies evolution using graph grammars, in: Knowledge Science, Engineering and Management, Springer, 2013, pp. 64–75.

[19] N. Stojanovic, L. Stojanovic, S. Handschuh, Evolution in the ontology-based knowledge management system, in: Proceedings of the European Conference on Information Systems-ECIS, 2002.

[20] P.-H. Luong, R. Dieng-Kuntz, A rule-based approach for semantic annotation evolution, Computational Intelligence 23 (3) (2007) 320–338.

[21] E. Sirin, B. Parsia, B. Grau, A. Kalyanpur, Y. Katz, Pellet: A practical owl-dl reasoner, Web Semantics: science, services and agents on the World Wide Web 5 (2) (2007) 51–53.

[22] M. Javed, Y. M. Abgaz, C. Pahl, Ontology change management and identification of change patterns, Journal on Data Semantics 2 (2-3) (2013) 119–143.

[23] M. Gueffaz, P. Pittet, S. Rampacek, C. Cruz, C. Nicolle, Inconsistency identification in dynamic ontologies based on model checking., in: WEBIST, SciTePress, 2012, pp. 418–421.

[24] L. Liu, P. Zhang, R. Fan, R. Zhang, H. Yang, Modeling ontology evolution with setpi, Information Sciences 255 (2014) 155–169.

[25] R. Milner, Communicating and mobile systems: the pi calculus, Cambridge university press, 1999.

[26] G. Rozenberg, Handbook of graph grammars and computing by graph transformation, Vol. 1, World Scientific, 1999.

[27] H. Ehrig, M. Pfender, H. J. Schneider, Graph-grammars: An algebraic approach, in: Switching and Automata Theory, 1973. SWAT'08. IEEE Conference Record of 14th Annual Symposium on, IEEE, 1973, pp. 167–180.

[28] M. Barr, C. Wells, Category theory for computing science, Vol. 10, Prentice Hall New York, 1990.

[29] M. Löwe, Algebraic approach to single-pushout graph transformation, Theoretical Computer Science 109 (1) (1993) 181–224.

[30] H. Ehrig, Introduction to the algebraic theory of graph grammars (a survey), in: Graph-Grammars and Their Application to Computer Science and Biology, Springer, 1979, pp. 1–69.

[31] I. Horrocks, U. Sattler, S. Tobies, Practical reasoning for very expressive description logics, Logic Journal of IGPL 8 (3) (2000) 239–263.

[32] N. F. Noy, M. Klein, Ontology evolution: Not the same as schema evolution, Knowledge and information systems 6 (4) (2004) 428–440.

[33] P. Haase, L. Stojanovic, Consistent evolution of owl ontologies, in: The Semantic Web: Research and Applications, Springer, 2005, pp. 182–197.

[34] L. Qin, V. Atluri, Evaluating the validity of data instances against ontology evolution over the semantic web, Information and Software Technology 51 (1) (2009) 83–97.

[35] C. Ermel., M. Rudolf., G. Taentzer, The agg approach: Language and environment, in: Handbook of graph grammars and computing by graph transformation, World Scientific Publishing Co., Inc., 1999, pp. 551–603.

[36] U. Nickel, J. Niere, A. Zündorf, The fujaba environment, in: Proceedings of the 22nd international conference on Software engineering, ACM, 2000, pp. 742–745.

[37] D. Varró, A. Pataricza, Generic and meta-transformations for model transformation engineering, 2004-The Unified Modeling Language. Modelling Languages and Applications (2004) 290–304.

[38] G. Karsai, A. Agrawal, F. Shi, J. Sprinkle, On the use of graph transformations in the formal specification of computer-based systems, in: Proceedings of IEEE TC-ECBS and IFIP10. 1 Joint Workshop on Formal Specifications of Computer-Based Systems, 2003, pp. 19–27.

[39] F. Fondement, P.-A. Muller, L. Thiry, B. Wittmann, G. Forestier, Big metamodels are evil, in: Model-Driven Engineering Languages and Systems, Springer, 2013, pp. 138–153.