Model-Driven Software Engineering in the Resource Description Framework: a way to version control

Hans Georg Schaathun* Adrian Rutle†

Abstract

Version control is a long-standing open problem for model-driven software engineering (MDSE). When the users work with diagrammatic models, conventional text-based version control is of little use, as the textual representation may change even when no change is made to the semantic model. A similar problem exists in semantic technologies, but recent research has given us a couple of very promising solutions for Resource Description Framework (RDF) graphs. In this paper we demonstrate how the Diagram Predicate Framework (DPF), a modelling framework in MDSE, can be represented unambiguously as RDF graphs. The RDF representation is then used as a foundation to analyse conflict resolution for version control of DPF models.

1 Introduction

Model-driven engineering (MDSE) [5] represents a paradigm shift in the software development – from being code-centric to become model-centric – in order to tackle the increasing complexity of software. MDSE aims to organise its main building blocks, i.e. models, metamodel, and model transformations, in a well-structured engineering methodology. There have been several successful industrial applications of MDSE with gains in terms of productivity, quality and performance. See [24] for a review of MDSE's current state and practise. As models become first-class entities in the development process, model management tools and technologies become crucial. A critical element of every software engineering tool-chain is version control, to support parallel and concurrent development branches which can later be compared and/or merged. Text-based version control systems are mature, and have proved effective for conventional programming. For MDSE it remains a hard problem [6, 17, 4].

The fundamental difference between MDSE and conventional programming is that the user in MDSE works with graphs rather than line-based text [7, 1]. A simple change in the graph structure, will often result in many changed lines spread throughout the serialised file. Hence using line-based version control for models corrupts the information in the graph-based structure and associated syntactic and semantic information might get lost. Version control systems should therefore take the graph structure into account when graph-based artifacts are handled.

*NTNU, Noregs Teknisk-Naturvitskaplege Universitet, ⟨georg@schaathun.net⟩

†Høgskulen på Vestlandet, ⟨Adrian.Rutle@hvl.no⟩

This paper was presented at the NIK-2018 conference; see http://www.nik.no/.

A second challenge in MDSE is that the models have a user-friendly visual syntax in addition to the textual, semantic syntax. Models are usually created using the visual syntax, and therefore a fine-grained version control is necessary to take care of both the abstract and the concrete syntax. It is important to distinguish between changes to the graph-structure and changes to the visual representation, since the latter affects only the presentation of the model and not its meaning.

Furthermore, merging different layout is still a challenging task since no concise definition is yet done telling which layout changes should be reported as conflicts and how these conflicts should be presented to the user. Consequently, the modifications and conflicts are usually presented in the abstract syntax without the visual information which is valuable for the human user. The visualization of differences and conflicts in the concrete syntax of the modeling language is still an open issue [7]. On the bright side, some proposals to an infrastructure supporting optimistic model versioning has started to emerge (see [16] for a survey).

A popular and generic language to represent graph structures is the Resource Description Framework (RDF). Significant progress has been made on version control for RDF graphs in recent years. Difficult problems, such as hashing and handling of blank nodes, have been solved with promising prototype implementations, even though no industry standard have emerged yet.

In this paper we consider the Diagram Predicate Framework (DPF) [20] which is one framework for MDSE. We chose DPF due to its expressive power which facilitates the formalisation of MOF-based modelling languages [21] like the industrial standards EMF (Eclipse Modeling Framework [22]) and UML class diagrams. Our objective is to establish a simple framework for version control, merging, and conflict resolution for DPF models, based on an RDF representation. We emphasise the simplicity of RDF, which provides a simple and flexible data model, whose benefits extend beyond version control. For instance, the reasoners made for ontologies may also be relevant for MDSE.

We make a quick review of the relevant state of the art in Section 2 and an introduction to DPF in Section 3. The RDF representation is presented as an example in Section 4 and formalised in Section 5. In Section 6 we discuss version control, and in Section 7, we conclude.

2 Literature review

In this section, we will first outline the current state of version control in MDSE, then present some results on applying version control in RDF and ontologies. Finally, we present some previous works which have attempted to make the link between MDSE and ontologies with the purpose of solving the version control problem.

Current approaches to merging two software models can be categorized along two orthogonal dimensions: the representation of the artifacts may be either text-based or graph-based, and the representation and merging of patches (deltas) can be either state-based or operation-based [7].

In [19] an operation- and graph-based approach to version control in MDSE is presented. While this approach also uses the DPF and formalises version control concepts in terms of categorical constructions, in this paper we design a practical implementation for supporting version control by serialising DPF specifications into RDF. EMF Store [16] provides a dedicated framework for version control of EMF models. It is an operation- and graph-based approach which supports efficient and

precise detection of composite changes. AMOR [8] is a state-, operation- and graph-based approach which implements a conflict detection component for EMF models which reports conflicts resulting from both atomic and composite changes.

The R43ples [12] system is a server-side system to handle revision history of RDF graphs. It includes an extension of SPARQL to query against specific reversions or difference sets. Selected versions are stored as full graphs, while most are represented by delta sets, and need to be recomputed using some full graph. Full graphs and delta sets are stored as named graphs, which means that named graphs cannot be used for other purposes. Frommhold *et al.* [11] make a comprehensive suggestion for RDF versioning, supporting both named graphs, blank nodes, and hashing of RDF graphs.

Neither [12] nor [11] considered merging and conflict resolution. Their work focused on efficient persistence and querying against the version history. In [9], revision history is modelled as a sequence of patches. A conflict arises when two parallel patches give different results depending on the order in which they are applied. Building on their work, Hensel *et al.* [13] proposed a methodology for conflict detection and resolution in a three-way merge for semantic revision control systems. They also implemented a prototype based on R43ples [12], extending SPARQL further to allow merging queries.

The Quit (quad in git) store [2] is based on a canonical serialisation of an RDF graph, using the n-quad format with quads sorted in lexicographic ordering. These canonical text files can then be managed by git.

Blank nodes present certain challenges in the processing of RDF graphs, and their interpretation is not consistent in published datasets [15]. Tumarello *et al.* [23] define Minimum Self-contained Graphs (MSG). Any graph may be partitioned into MSGs so that every blank node is present in only one MSG and every statement occurs in exactly one MSG. As long as the MSGs are relatively small, isomorphism checks are tractable in practice.

Needless to say, MDSE and ontologies are related in many aspects [18, 3]; for example both involve domain modelling, (meta)modelling hierarchies, etc. But in this section we only present the works/tools which attempt to provide solution for serialising EMF models in RDF. We evaluate these approaches as potential tools for transforming our DPF specifications (which are internally represented as EMF models) into RDF.

Hillairet *et al.* [14] discussed a bridging between EMF and RDF. They define a mapping between EMF models (instances of Ecore) and OWL/RDFS ontologies. Moreover, they define a serialization mechanism for instances of these EMF models into RDF Resources. The mapping definition is implemented as an ATL transformation which can be applied to instances of the EMF model in order to generate the RDF Resources. This transformation is defined to be bidirectional, i.e., it can be used for both serialisation and deserialisation of the models. The authors also utilise their approach with query languages like HQL and SPARQL. EMF Triple[1] is a tool for serialising EMF models in RDF. It supports various RDF stores, such as Jena TDB and sparql endpoints. It also supports various graph databases such as Neo4J, through the sail implementations provided by tinkerpop blueprints.

[1] `https://marketplace.eclipse.org/content/emf-triple`

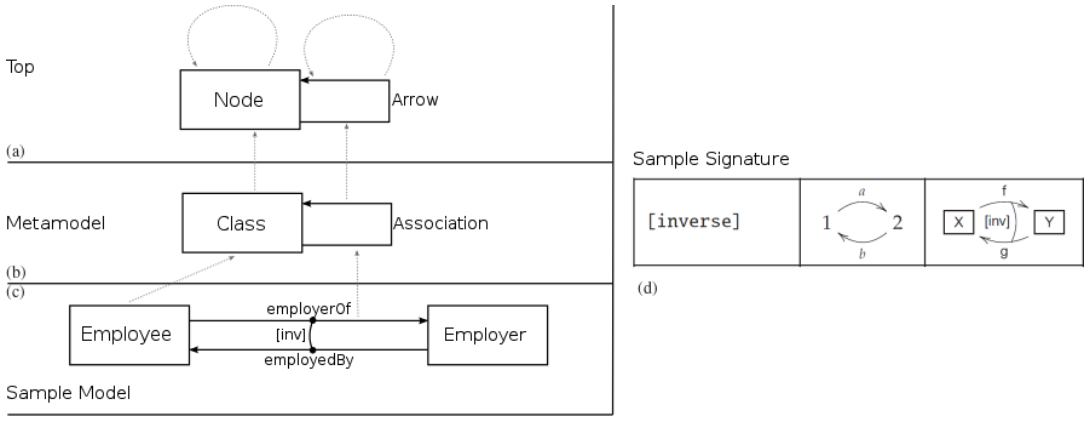

Figure 1: A metamodelling hierachy in DPF

3 Diagram Predicate Framework

DPF [20] provides a formal diagrammatic approach to (meta)modelling and model transformation based on category theory. The main concepts in DPF are specifications and signatures. A specification consists of a graph (which represents a model or metamodel) and constraints (which are used to specify domain peculiarities). Signatures define the diagrammatic predicates which are used to define the constraints. Formally, we define these concepts as follows.

Definition 1 (Signature). *A (diagrammatic predicate) signature* $\Sigma = (P^\Sigma, \alpha^\Sigma)$ *consists of a collection of predicate symbols* P^Σ *with a map* α^Σ *that assigns a graph to each predicate symbol* $p \in P^\Sigma$. $\alpha^\Sigma(p)$ *is called the arity of the predicate symbol* p.

Definition 2 (Atomic Constraint). *Given a signature* $\Sigma = (P^\Sigma, \alpha^\Sigma)$, *an atomic constraint* (p, δ) *added to a graph* S *is given by a predicate symbol* p *and a graph homomorphism* $\delta : \alpha^\Sigma(p) \to S$.

Definition 3 (Specification). *Given a signature* $\Sigma = (P^\Sigma, \alpha^\Sigma)$, *a (diagrammatic) specification* $\mathfrak{S} = (S, C^\mathfrak{S} : \Sigma)$ *is given by a graph* S *and a set* $C^\mathfrak{S}$ *of atomic constraints* (p, δ) *on* S *with* $p \in P^\Sigma$.

An example specification is shown in Figure 1(c). In addition to the graph, with two nodes and two arrows, the specification includes a set of constraints. In this case there is one constraint, saying that the two arrows are inverses of each other. The signature (Figure 1(d)) serves as a metamodel for constraints, defining the interface used to apply a constraint to an actual model.

In most cases, it is necessary to define the types of model elements. These types and their relations are defined in a metamodel, which in itself is a specification. In the example, the metamodel is shown in Figure 1(b). The type mapping is shown as dotted arrows, and it defines a graph homomorphism from the typed graph to the type graph. This idea of typing leads to an organisation of models in metamodelling stacks, where a model at a level acts as the metamodel for the model below it. The top-level model (Figure 1(a)) is its own metamodel, and we can see the typing homomorphism forming self loops. In DPF, each of these models is represented as a typed specification, which is an ordinary specification together with a graph homomorphism ι. Formally we define it as follows.

Listing 1 Proposed Turtle syntax for the top level artefact (Fig. 1(a)).

```
@prefix mm0:  <http://example.org/mm0.ttl#> .
<http://example.org/mm0.ttl> rdf:type dpf:Specification ;
       dc:creator  "Adrian Rutle" ;
       rdfs:label  "Top-level metamodel" .
mm0:node rdf:type mm0:node .
mm0:edge rdf:type mm0:edge .
mm0:node mm0:edge mm0:node .
```

Listing 2 Sample signature file as RDF (Fig. 1(d)).

```
<http://example.org/samplesig.ttl> a dpf:Signature .
dpf:inverse a dpf:PredicateSymbol ;
                dpf:hasNode :a, :b ;
                dpf:hasEdge :f, :g .
:a :f :b .
:b :g :a .

dpf:composition a dpf:PredicateSymbol ;
                dpf:hasNode :c, :d, :e ;
                dpf:hasEdge :h, :i, :j .
:c :h :d .
:d :i :e .
:c :j :e .
```

Definition 4 (Typed Specification). *Given a specification* $\mathfrak{T} = (T, C^{\mathfrak{T}} : \Sigma^T)$*, a typed specification* $\mathfrak{S} \triangleright \mathfrak{T} = (S, C^{\mathfrak{S}} : \Sigma^S)$ *over* $\mathfrak{T}$ *is a specification along with a graph homomorphism* $\iota^S : S \to T$*.*

4 DPF as RDF by example

For the sake of presentation, we take the model stack in Figure 1 as an example, and illustrate, step by step, how we could make an RDF representation thereof. In the next section, we will formalise the approach, and argue that it is complete.

The top-level metamodel in DPF (Figure 1(a)), can be represented in RDF (Turtle syntax) as follows:

```
:node :arrow :node .
:node rdf:type :node .
:arrow rdf:type :arrow .
```

The first line defines the graph with a single arrow, and the other two give the type information. The artifact should have a unique ID, and may have associated metadata. Adding this, we get Listing 1. The triple which specifies the type `dpf:Specification` implicitly refers to the artifact as represented by the file as a whole, and `dpf:mm0` becomes the URI for this artifact. Note that this pattern is similar to how modular ontologies are defined in OWL.

A signature is a set of elements, where each signature element corresponds to a predicate label and serves as a template for a constraint. The signature elements can

Listing 3 A sample DPF specification coded in RDF (Fig. 1(c)).

```
@prefix :  <http://example.org/samplemodel.ttl#> .
@prefix mm1:  <http://example.org/mm1.ttl#> .
@prefix sig:  <http://example.org/samplesig.ttl#> .

<http://example.org/samplemodel.ttl>
   rdf:type dpf:Specification ;
   dpf:import <http://example.org/mm1.ttl> ;
   dpf:import <http://example.org/samplesig.ttl> ;
   rdfs:label  "Sample model" .

:employer rdf:type mm1:class .
:employee rdf:type mm1:class .
:employedBy rdf:type mm1:association .
:employer :employedBy :employee .
:employerOf rdf:type mm1:association .
:employee :employerOf :employer .

:delta1 dpf:mapForPredicate dpf:inverse .
sig:a   :delta1   :employer   .
sig:b   :delta1   :employee   .
sig:f   :delta1   :employedBy .
sig:g   :delta1   :employerOf .
```

be handled individually. Many common signature elements are already supported by OWL, for instance `owl:inverseOf`. DPF also requires signature elements of higher arity, for instance composition which says that one edge f may be the composition $f = g \circ h$ of two other edges.

To handle arbitrary arity, we propose an RDF representation which closely follows Definition 1. An example is shown in Listing 2. The signature has a URI as for the model graphs before. This signature defines two predicate symbols `dpf:inverse` and `dpf:composition`. Each has a set of nodes and edges which make up the arity graph. For instance α(`dpf:inverse`) has two nodes and two edges. The following two lines in Listing 2 give the graph, where both edges connect the two nodes, but in opposite directions.

Listing 3 shows a second artifact, representing the metamodel one level below. Note the new property `dpf:import`, which is used to import both the metamodel and the signature. The interpretation of `dpf:import` is similar to, possibly identical to, `owl:import`, but we have introduced a specific DPF term to be able to restrict the domain and range to DPF artifacts.

The last five lines give the constraint (`dpf : inverse`, δ_1). We assert that `:delta1` is a mapping δ_1 for the predicate `dpf:inverse`. Then we assert the image under δ_1 for each element of the graph associated with `dpf:inverse`.

5 Formalising the RDF representation

We have introduced a vocabulary of artifacts, defining the artifact types `dpf:Signature` and `dpf:Specification`, and `dpf:import` to define dependencies between artifacts. These two types of artifacts are sufficient for practical use. Signatures provide the language for constraints, and (typed) specifications are used for models including metamodels. In RDF, we represent each artifact as a named graph, with a URI. Thus the artifacts can be handled separately, and dependencies and other metadata can be defined.

Definition of RDF

Definition 5. *An RDF document is a set of triples* $(s, p, o) \in (U \cup B) \times U \times (U \cup B \cup D)$*, where* U *is a set of URIs (universally unique identifiers),* B *is a set of blind nodes (which can be viewed as local identifiers), and* D *is a set of literals.*

In the example, we used the set U for all model elements. The literals L were used for some of the metadata (strings). Blind nodes are not needed, and we shall disregard the set B.

An RDF document is often called an RDF graph, where we view the triple (s, p, o) as an arrow (edge) from vertex s to vertex o labelled by p. RDF has a few unorthodox properties when viewed as graph. Firstly, the same element may be both a vertex and an edge label. Secondly, some nodes, namely the set of literals D, can only appear as a destination node of an arrow. Finally, the set of vertices V is not explicitly defined, and a vertex can only exist if it is connected to some arrow. The triple (s, p, o) can also be viewed as a logic statement, with a subject s, predicate p, and an object o. We will rely on the graph interpretation in this paper.

Many different ontology languages use RDF as their representation language. RDF itself does not have any semantics, making it versatile and reusable. We use a few terms from the RDF, RDFS, and DC (Dublin Core) vocabularies, and use their canonical prefixes `rdf:`, `rdfs:`, and `dc:` without further comment.

Identifiers

The use of identifiers in DPF has not been standardised, and different implementations have handled it differently. This prevents serialised formats to be shared between implementations, so it is a problem which has to be resolved at some point. In practice, we will need to refer unambiguosly to every resource in a DPF model, including the artifacts, nodes, edges, and predicate symbols. In an RDF representation, the universal ID should be a URI. If required, a mapping can be designed between URIs and other global naming schemes, such as the package naming scheme used in Eclipse and in Java. This may be useful for the artifacts. For other resources, we suggest to use the artifact URI as a prefix and append a local ID, as illustrated in Listings 1 and 3.

The local ID can be machine- or user-generated, but the user-generated ID is likely to change. A machine-generated ID makes it easier to make it unique and immutable, and tools may hide this ID from the user. A user-defined label must be supported in addition to the ID, and this can be stored as a separete triple, with the `rdfs:label` annotation property.

Signatures

The vocabulary for signatures requires three terms: `dpf:PredicateSymbol`, `dpf:hasNode`, and `dpf:hasEdge`. This is sufficient to define a signature $\Sigma = (P^\Sigma, \alpha^\Sigma)$ according to Definition 1. The set P^Σ consists of all the URIs asserted with type `dpf:PredicateSymbol`. The associated graph $\alpha^\Sigma(p)$ for a given predicate symbol p, consists of the nodes and edges associated with p via the `dpf:hasNode` and `dpf:hasEdge` properties. The graph itself, is simply represented as an RDF graph, giving a (source,edge,destination) triple for each edge. Any signature can be represented in RDF in this way.

Specifications

A typed specification consists of three parts: the model graph, the set of constraints, and the typing. Being just a directed multigraph, the model graph can be represented as RDF in the natural way. We use the singleton pattern, so that a new property p is defined for every edge (s, p, o) and used only once. Thus p can be used to identify the edge uniquely.

The typing is a homomorphism from the underlying graph of a specification to the type graph. The homomorphism can trivially be represented as a bipartite graph, which we define in RDF using the `rdf:type` property to connect corresponding nodes and edges.

Each constraint (p, δ) is a predicate label p and a homomorphism δ from the graph associated with p to a subgraph of the underlying graph. We have introduced the term `dpf:mapForPredicate` to declare the constraint (p, δ). Again the homomorphism is modelled as a bipartite graph.

Visual syntax

DPF has a visual syntax, where geometric layout must be serialised in addition to the semantic content. The dual use of visual and textual syntax is also known from other languages, such as Modelica [10], where the visualisation is specified by optional annotations in the textual program. In the RDF representation, we suggest to keep the visualisation in a separate artifact, as is done in the current Eclipse plugin for DPF. This separation reduces coupling between the semantic model and its presentation. We will leave the vocabulary for visualisation for future work, and just note that the visualisation artifact can make declare geometric coordinates for the different components.

6 Classification and resolution of conflicts

Able to represent DPF as RDF graphs, we can use any of the existing serialisation formats and tools for RDF to process DPF models, e.g. R43ples or Quit as mentioned in the literature review. These tools do not handle conflict resolution. In this section with investigate what constitutes a conflict and how it can be resolved.

Patch representation in RDF

An RDF document is a set V_0 of triples. If we create a new document V_1 as a revision of V_0, we can define the patch as the pair $(V_1 \backslash V_0, V_0 \backslash V_1)$ where $\backslash$ denotes the set difference. The set $V_1 \backslash V_0$ is the triples to be added and $V_0 \backslash V_1$ the set of triples to be removed by the patch.

We consider state-based, three-way merging. To merge two version V_a and V_b, we need to identify the last common ancestor V. Since all versions are ultimately derived from the same (empty) initial version, such a last common version always exists. We can calculate the patches for each version $P_a = (V_a \backslash V, V \backslash V_a)$ and $P_b = (V_b \backslash V, V \backslash V_b)$, and define the merged patch $P_c = ((V_a \backslash V) \cup (V_b \backslash V), (V \backslash V_a) \cup (V \backslash V_b))$. The merged version V_c is defined by applying the patch P_c to V.

Definition 6. *Assuming that V_a and V_b are two versions representing valid models, there is a* conflict *between V_a and V_b if the merged version V_c is not a valid model.*

Any set of triples is a valid RDF document. Hence, conflicts occur only when semantic constraints are added on top of RDF. This definition of conflicts is different from text-based version control, where conflicts occur when changes are made to adjacent lines. It is also different from conflicts considered for RDF in [9, 13], where every intermediate change in the history from V to V_a (resp. V_b) is considered.

Syntactic conflicts in DPF

There is no agreement in the literature on the definition of syntactic conflicts [1]. Syntax is defined by the language used, and in metamodelling we use a stack of languages (see Fig. 1), each with its own (abstract) syntax. As syntactic conflicts we shall only count such conflicts as can be identified with the languages that we have introduced. Merging two typed specifications to get a merged specification $\mathfrak{S} = (S, C^{\mathfrak{S}} : \Sigma)$, the following syntactic conflicts are possible:

1. Violation of the mathematical language of graphs, i.e. S is not a graph.
2. Violation of the syntax for definition of constraints, i.e. there is a constraint $(p, \delta) \in C^{\mathfrak{S}}$ so that δ is not a graph homomorphism.
3. Type violation, i.e. ι is not a graph homomorphism.

It is straight forward to identify these conflicts by checking the consistency of the logic system represented by the RDF graph.

The prototypical example of (1) is when one version removes a node, while the other adds new references to the same node (edges to or from the node). Type (2) arises if one version removes a model element, while the other adds a constraint on that particular element. Type violations (3) can occur if one branch changes the typing and the other changes the nodes and edges involved. Checking this automatically is normally possible.

Removal of a node is not well-defined in RDF, where a node exists if and only if it is mentioned in some triple. In DPF, this situation is still detectible because it results in a node or arrow with no type statement. If we want to support untyped nodes and edges in DPF, this can be resolved with a dummy type (e.g. `dpf:unknownType`), to allow a type statement with unknown type.

Syntactic conflicts can easily be detected by checking the RDF subgraphs against the definitions of graphs and graph homomorphisms. In most cases, they will require manual resolution, so tools should focus on identifying and locating the conflicts.

Merging two versions of a signature results in a conflict if the merged version does not satisfy Definition 1. This is similar to (1) above in the sense of creating an arity which is not a valid graph. Since the signatures do not have a metamodel, the other syntactic conflicts do not apply.

Textual conflicts

Textual conflicts are only visible in the textual representation. The two versions, in presence of textual conflicts, represent the same or equivalent graphs. In RDF this can take three forms, as a change to a URI, to a blank node, or to a literal.

A typical example is renaming of a model element, i.e. the user-defined label (`rdfs:label`) changes. A conflict occurs when two users both change the label on the same model element. If a model element is allowed to have multiple labels, there is no conflict. A conflict only arises if the property p only allows one statement (s, p, o) for the same subject s. Such cardinality constraints may be defined for each p and checked semantically after merging. Changes to the metadata (e.g. author name) may also cause textual conflicts. If a textual conflict is detected, it must be presented to the user for manual resolution.

Blank nodes are not used, but a URI conflict occurs if both users add the same new subgraph while the tools create different URIs. Identical changes may be detected using isomorphism checks on the patches, thus identifying it as a textual conflict. It is questionable how useful this is in practice. Two users are very unlikely to create identical changes, especially complex changes. More commonly, they choose slightly different variations, even if they have the same intention, and the two versions need to be compared and evaluated manually.

Semantic and other conflicts

Semantic conflicts is a diverse range [1], and we can only briefly mention a few examples. A conflict occurs if the merged model has no possible instantiation. Although computationally expensive, this can be checked automatically. Other conflicts arise when two users add different model components with the same intention. Such conflicts cannot necessarily by automatically detected, and resolution will often require domain knowledge.

Conformance violation occur when S violates constraints defined in the metamodel. This is a syntactic conflict, but it depends on the semantics of the predicates being defined, and different languages are needed to identify the conflicts.

A last class of conflicts can occur if the metamodel or signature is changed, i.e. when we merge two versions importing different documents to define the same resources. In such cases, one of the models should probably be updated to conform with the same metamodel and signature as the other, before the merge is attempted.

Visualising conflicts

Although many conflicts can be detected automatically, resolution often requires manual intervention. This can be facilitated if the software can present the conflict in an intelligible way. A main criticism towards text-based version control in the MDSE community is that a simple changes, such as the removal of a node, may affect many lines of code scattered around the document. Thus the conflict is hard to comprehend. This problem is not unique to modelling, and it occurs with refactoring in conventional programming, for instance when a function is renamed.

Text-based tools will usually flag a conflict if changes occur within adjacent lines. The analogue in RDF would be changes to adjacent nodes and edges. To visualise conflicts, we should therefore focus on changing edges which are connected in a graph-theoretical sense. The subgraph which is affected by the merge, can be

identified by taking every edge in the merged patch:

$$\Delta = (V_a \backslash V) \cup (V \backslash V_a) \cup (V_b \backslash V) \cup (V \backslash V_b).$$

This is a set of edges and thus a graph, not necessarily connected. By partitioning this graph into connected components, we identify changes which are likely to relate to the same conflict, and which should therefore be presented together.

7 Conclusion

We have provided a proof of concept to demonstrate that DPF can be serialised in RDF. This provides a simple and flexible representation with human readable serialisations. The extensive literature and wide applications on RDF means that it is well understood. Since most MDSE frameworks are based on graphs and DPF can represent graph-based modeling languages, it is reasonable to believe that the work can be extended to beyond the single framework of DPF.

Based on the RDF representation we discuss how merge conflicts in a version control system can be categorised and resolved. Considering state-based, three-way merging, conflicts only occur as violations of DPF semantics. No conflicts exist in the RDF representation itself. Hence, existing RDF tools only can be used to manage revision history. In contrast, conflict resolution must be implement on a higher level. We have categorised potential conflicts and outlined approaches to resolution.

We are confident that this provides a working solution, and the most immediate future work is to make a prototype implementation, both of the RDF serialisation of DPF, and of the conflict resolution for DPF version control based on RDF. Some open questions remain to optimise the solution, including using model constraints to automate as much as possible of the conflict resolution, and to provide a suitable framework for release and dependency management.

The RDF representation gives access to a rich toolbox, already developed in the RDF and semantic web communities, and the benefits may extend beyond version control. For instance, automatic reasoning developed for ontologies may find applications in MDSE.

References

[1] K Altmanninger and Alfonso Pierantonio. A categorization for conflicts in model versioning. *e & i Elektrotechnik und Informationstechnik*, 128(11-12):421–426, 2011.

[2] Natanael Arndt, Norman Radtke, and Michael Martin. Distributed collaboration on rdf datasets using git: Towards the quit store. In *Proceedings of the 12th International Conference on Semantic Systems*, pages 25–32. ACM, 2016.

[3] Uwe Aßmann, Steffen Zschaler, and Gerd Wagner. Ontologies, meta-models, and the model-driven paradigm. In Coral Calero, Francisco Ruiz, and Mario Piattini, editors, *Ontologies for Software Engineering and Software Technology*, pages 249–273. Springer, 2006.

[4] Stephen Barrett, Patrice Chalin, and Greg Butler. Model merging falls short of software engineering needs. In *Proceedings of the 2nd Workshop on Model-Driven Software Evolution @MoDELS'08*, 01 2008.

[5] Marco Brambilla, Jordi Cabot, and Manuel Wimmer. *Model-Driven Software Engineering in Practice, Second Edition*. Synthesis Lectures on Software Engineering. Morgan & Claypool Publishers, 2017.

[6] Petra Brosch, Gerti Kappel, Philip Langer, Martina Seidl, Konrad Wieland, and Manuel Wimmer. An introduction to model versioning. In Marco Bernardo, Vittorio Cortellessa, and Alfonso Pierantonio, editors, *Formal Methods for Model-Driven Engineering*, volume 7320 of *LNCS*, pages 336–398. Springer, 2012.

[7] Petra Brosch, Philip Langer, Martina Seidl, Konrad Wieland, Manuel Wimmer, and Gerti Kappel. The past, present, and future of model versioning. In *Emerging Technologies for the Evolution and Maintenance of Software Models*, pages 410–443. IGI Global, 2012.

[8] Petra Brosch, Martina Seidl, Konrad Wieland, Manuel Wimmer, and Philip Langer. By-example adaptation of the generic model versioning system AMOR: how to include language-specific features for improving the check-in process. In Shail Arora and Gary T. Leavens, editors, *OOPSLA*, pages 739–740. ACM, 2009.

[9] Steve Cassidy and James Ballantine. Version control for RDF triple stores. *ICSOFT (ISDM/EHST/DC)*, 7:5–12, 2007.

[10] Peter Fritzson. *Principles of object-oriented modeling and simulation with Modelica 2.1*. John Wiley & Sons, 2010.

[11] Marvin Frommhold, Rubén Navarro Piris, Natanael Arndt, Sebastian Tramp, Niklas Petersen, and Michael Martin. Towards versioning of arbitrary RDF data. In *Proceedings of the 12th International Conference on Semantic Systems*, pages 33–40. ACM, 2016.

[12] Markus Graube, Stephan Hensel, and Leon Urbas. R43ples: Revisions for triples. In *Proceedings of the 1st Workshop on Linked Data Quality co-located with 10th International Conference on Semantic Systems (SEMANTiCS 2014)*, 2014.

[13] Stephan Hensel, Markus Graube, and Leon Urbas. Methodology for conflict detection and resolution in semantic revision control systems. 2016.

[14] Guillaume Hillairet, Frédéric Bertrand, Jean Yves Lafaye, et al. Bridging emf applications and rdf data sources. In *Proceedings of the 4th International Workshop on Semantic Web Enabled Software Engineering, SWESE*, 2008.

[15] Aidan Hogan, Marcelo Arenas, Alejandro Mallea, and Axel Polleres. Everything you always wanted to know about blank nodes. *Web Semantics: Science, Services and Agents on the World Wide Web*, 27:42–69, 2014.

[16] Maximilian Koegel and Jonas Helming. Emfstore: a model repository for EMF models. In Jeff Kramer, Judith Bishop, Premkumar T. Devanbu, and Sebastián Uchitel, editors, *Proceedings of ICSE - Volume 2*, pages 307–308. ACM, 2010.

[17] Philip Langer. Version control for models: From research to industry and back again. In Tanja Mayerhofer, Alfonso Pierantonio, Bernhard Schätz, and Dalila Tamzalit, editors, *MODELS*, volume 1706 of *CEUR Workshop Proceedings*, page 1. CEUR-WS.org, 2016.

[18] Fernando Silva Parreiras, Jeff Z. Pan, and Uwe Aßmann. Second workshop on transforming and weaving ontologies in model driven engineering (TWOMDE 2009). In Sudipto Ghosh, editor, *MODELS*, volume 6002 of *LNCS*, pages 325–328. Springer, 2009.

[19] Alessandro Rossini, Adrian Rutle, Yngve Lamo, and Uwe Wolter. A formalisation of the copy-modify-merge approach to version control in MDE. *JLAMP*, 79(7):636–658, 2010.

[20] Adrian Rutle. *Diagram Predicate Framework: A Formal Approach to MDE*. PhD thesis, Department of Informatics, University of Bergen, November 2010.

[21] Adrian Rutle, Alessandro Rossini, Yngve Lamo, and Uwe Wolter. A diagrammatic formalisation of mof-based modelling languages. In Manuel Oriol and Bertrand Meyer, editors, *Objects, Components, Models and Patterns*, pages 37–56. Springer Berlin Heidelberg, 2009.

[22] Dave Steinberg, Frank Budinsky, Marcelo Paternostro, and Ed Merks. *EMF: Eclipse Modeling Framework (2nd Edition)*. Addison-Wesley Professional, 2008.

[23] Giovanni Tummarello, Christian Morbidoni, Paolo Puliti, and Francesco Piazza. Signing individual fragments of an RDF graph. In *Special interest tracks and posters of the 14th international conference on World Wide Web*, pages 1020–1021. ACM, 2005.

[24] Jon Whittle, John Hutchinson, and Mark Rouncefield. The state of practice in model-driven engineering. *IEEE Software*, 31(3):79–85, 2014.