https://doi.org/10.1007/s00165-021-00557-0

Formal Aspects of Computing (2021) 33: 1115–1145

Formal Aspects of Computing

Schema compliant consistency management via triple graph grammars and integer linear programming

Nils Weidmann[1] and Anthony Anjorin[2]

[1]Paderborn University, Paderborn, Germany
[2]IAV GmbH Ingenieurgesellschaft Auto und Verkehr, Berlin, Germany

Abstract. In the field of Model-Driven Engineering, Triple Graph Grammars (TGGs) play an important role as a rule-based means of implementing consistency management. From a declarative specification of a consistency relation, several operations including forward and backward transformations, (concurrent) synchronisation, and consistency checks can be automatically derived. For TGGs to be applicable in realistic application scenarios, expressiveness in terms of supported language features is very important. A TGG tool is schema compliant if it can take domain constraints, such as multiplicity constraints in a meta-model, into account when performing consistency management tasks. To guarantee schema compliance, most TGG tools allow application conditions to be attached as necessary to relevant rules. This strategy is problematic for at least two reasons: First, ensuring compliance to a sufficiently expressive schema for all previously mentioned derived operations is still an open challenge; to the best of our knowledge, all existing TGG tools only support a very restricted subset of application conditions. Second, it is conceptually demanding for the user to indirectly specify domain constraints as application conditions, especially because this has to be completely revisited every time the TGG or domain constraint is changed. While domain constraints can in theory be automatically transformed to obtain the required set of application conditions, this has only been successfully transferred to TGGs for a very limited subset of domain constraints. To address these limitations, this paper proposes a search-based strategy for achieving schema compliance. We show that all correctness and completeness properties, previously proven in a setting without domain constraints, still hold when schema compliance is to be additionally guaranteed. An implementation and experimental evaluation are provided to support our claim of practical applicability.

Keywords: Model-driven engineering, Triple graph grammars, Integer linear programming, Graph constraints, Application conditions

1. Introduction

Model transformation is an essential part of Model-Driven Engineering (MDE), as it is an essential means of keeping semantically interrelated models consistent. This requires the consistent transformation of models from one domain to another, the propagation of updates from one model to others, and the checking of all involved models for intra- and inter-model consistency. All these tasks, which we denote collectively as consistency management, are especially important in collaborative scenarios, where multiple (teams of) domain experts model systems with domain-specific languages (DSLs) tailored to their particular problem domain.

Correspondence to: Nils Weidmann, e-mail: nils.weidmann@upb.de

There are numerous application areas for consistency management techniques, including the industry automation domain [AYL+18], the automotive domain [GHN10], and language editor development [BPD+14].

As a special case of model transformation, bidirectional transformation (bx) addresses the problems posed by redundant implementations of the specified consistency relation for different operations: Instead of developing and maintaining separate transformations for forward and backward transformation, consistency checks, etc., all or at least some operations are derived from the underlying consistency relation or from other operations. Diverse approaches to bx have been proposed in different research communities, including functional programming, databases, and MDE. As a representative in the MDE domain, Triple Graph Grammars (TGGs) are a declarative and rule-based approach to bx based on algebraic graph transformation [Sch94]. To be suitable for real-world use cases, a transformation language has to be sufficiently expressive; increasing expressiveness while still guaranteeing all formal properties is an open challenge for ongoing research on TGGs [ALS15, WOR19]. In this paper, we propose an approach with which the fulfilment of domain constraints by TGG-based consistency management operations can be guaranteed. We shall denote this property in the rest of the paper as schema compliance.

Domain constraints can be formalised as graph constraints and include negative constraints, forbidding certain situations, positive constraints, demanding certain patterns, and more general constraints enforcing implications between graph patterns (cf. Ehrig et al. [EEPT06] for a general introduction to graph constraints). A well-known example for domain constraints are multiplicity constraints specified in the metamodels of the respective domains. Suppose that for an association, a multiplicity of `m..n` shall be respected. The upper bound can be guaranteed by forbidding `n+1` occurrences of the respective element. In turn, the lower bound of `n` can be demanded with an implication constraint (as soon as such an association exists, there must be a match for `n` elements). While numerous TGG tools have been implemented [HLG+13], we are not aware of any tool that provides direct and extensive support for schema compliance. Most TGG tools only allow the user to introduce constraints indirectly, by attaching application conditions (ACs) to rules to restrict their applicability. This is challenging for the user, especially for large TGGs, as all rules must be analysed carefully after every single change. There has been a proposal to support schema compliance by automatically transforming domain constraints to ACs [AST12], but only a very small subset of Negative Application Conditions (NACs) (and consequently negative constraints), is supported. Finally, the focus has been primarily on forward/backward transformation and synchronisation, and less on other operations such as consistency checking or concurrent synchronisation [WFA20].

The approach presented in this paper builds upon seminal work on combining TGGs with Integer Linear Programming (ILP) for consistency management. This hybrid strategy combines a generic and flexible problem definition with acceptable runtime behaviour for growing model sizes. Consistency is no longer seen as a binary property: If perfect consistency cannot be achieved, an "optimal" partial solution is computed which, e.g., maximises the number of consistently transformed elements. After proposing this strategy for consistency checking [Leb16, LAS17] and providing proofs for correctness and completeness [Leb18], the approach was generalised to cover unidirectional transformations [WALS19]. The primary advantage of this hybrid TGG- and search-based consistency management approach is that all derived operations can be uniformly handled, differing only in which models are provided, and which are to be produced. Furthermore, all operations are flexible enough to handle inconsistent input, computing a partial result instead of simply rejecting such input as invalid. In a conference version of this paper [WA20], we extended the approach to support schema compliance (up to implications of graph patterns) for the consistency checking operation.

In this paper, we now provide an extension of this formal framework (schema compliance also up to implications), to cover forward and backward transformation, and correspondence link creation. We show that correctness and completeness guarantees can still be provided for these additional operations, even though they now create new elements as part of the consistency restoration process. Due to the flexibility of our search-based approach, we take a further step towards fault-tolerant consistency management as all supported operations terminate with a maximal partial solution that is contained in the language of the underlying TGG and respects all posed domain constraints. Conventional TGG-based approaches often separate checking domain constraints from the transformation, such that the user is forced to fix all constraint violations before the actual transformation task can be started.

The rest of the paper is structured as follows: A running example is presented in Sect. 2, which is used to provide an intuitive introduction to the approach based on a small backward transformation example in Sect. 3. After a brief introduction to the foundations of algebraic graph transformation (Sect. 4), our approach is formalised in Sect. 5, so that correctness and completeness can be shown in Sect. 6. An implementation in an actively developed TGG tool is presented in Sect. 7, followed by an experimental runtime evaluation in Sect. 8. Our contribution is compared with related work in Sect. 9, before Sect. 10 concludes the paper, and gives an outlook on possible future work.

2. Running example

To demonstrate our approach, we consider a consistency relation between (simplified) abstract syntax trees describing Java code (source model) and its documentation (target model). The respective metamodels are depicted in Fig. 2. As the root of the Java metamodel, `Classes` can form an inheritance hierarchy. Each class can have arbitrarily many `Methods` and `Fields`, and each `Method` can have a set of `Parameters`. The documentation metamodel, in contrast, consists of `Documents` that can reference each other via hyper-references. A document is structured as a set of `Entries`. To provide an overview of important terms, a `Glossary` is contained in the documentation model. It consists of `GlossaryEntries`, which are referred to from documentation entries. The correspondence relation is represented by a third *correspondence* model, which is depicted in form of diamonds between source and target models. Classes are associated with documents, while methods, their parameters, and fields are represented as entries in the respective document. The glossary and its entries do not have a corresponding structural element in the source model, therefore they are not linked to a node of the correspondence model. Besides conformance to the metamodels depicted in Fig. 2, we restrict the set of consistent triples for our running example by requiring three additional graph constraints to be satisfied (Fig. 2):

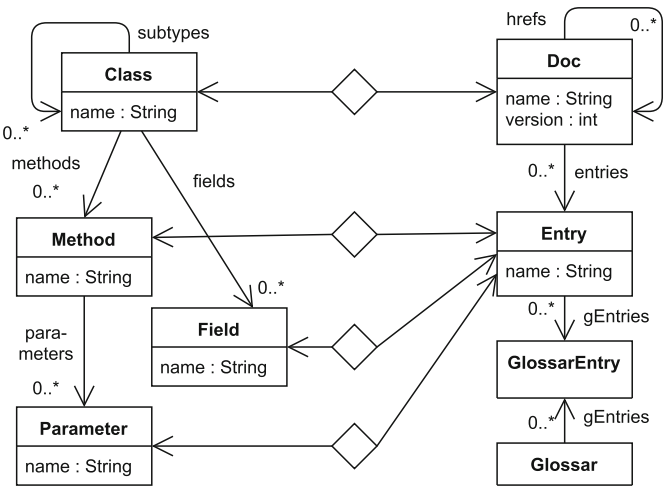


Fig. 1. Triple of Metamodels

- We *forbid* that there are two or more `Glossaries` in the documentation model with the constraint `NoTwoGlossaries`. This is a *negative constraint*.
- Creating multiple links from an `Entry` to a `GlossaryEntry` is forbidden by the constraint `NoDoubleLink`.
- In the Java model, we enforce that every class be non-empty. To specify this, we use an `implication` constraint expressing that each `Class` (premise) be connected to either a `Method` (Conclusion 1) or `Field` (Conclusion 2), forming the constraint `NoEmptyClass`.
- Implication constraints can be more complex and affect multiple models. With a fourth constraint `SameName-SameGlossaryEntry`, we ensure that methods of the same class with the same name (overloaded methods), correspond to entries in the documentation model that point to a common glossary entry.

3. Main ideas

Before formalising our approach, an intuitive explanation of how a documentation model is transformed to the Java domain is provided in this section. This "backward" transformation strategy is derived from declarative rules and guarantees that the previously presented graph constraints (Fig. 2) are respected. As the TGG approach is symmetric (source and target domains are interchangeable), the forward transformation works analogously.

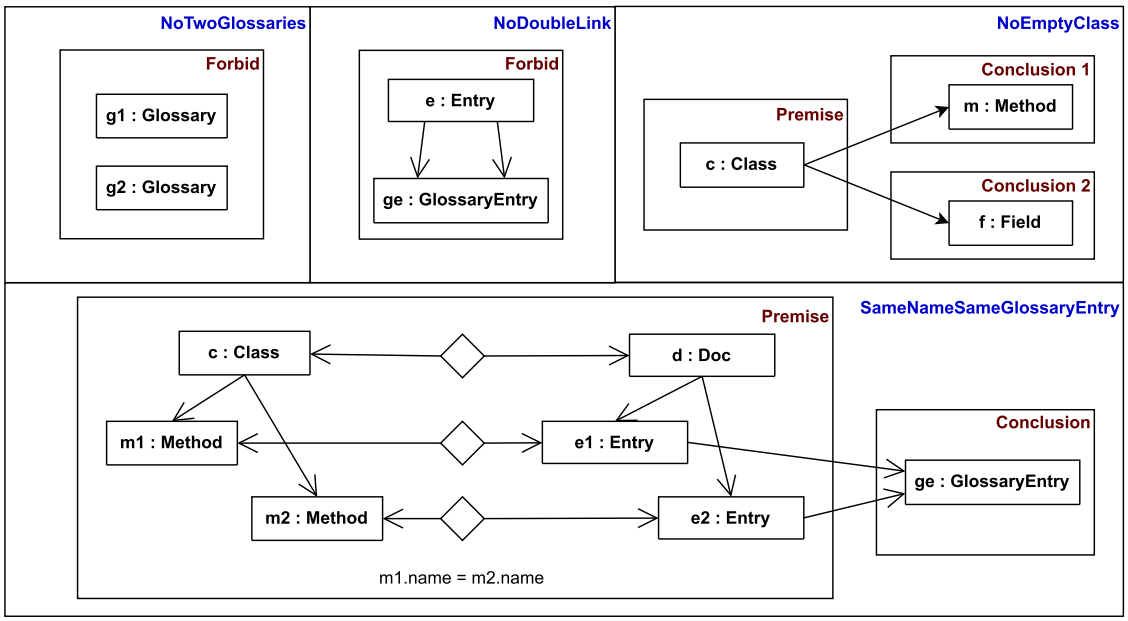


Fig. 2. Graph constraints for the TGG `JavaToDoc`

Introduction to TGGs: In order to define the consistency relation from which the backward transformation rule is derived, a set of six declarative TGG rules is used, which is depicted in Fig. 3. Nodes and edges are either coloured green (with a ++ markup) or black (without any additional markup). Green elements are created when applying the rule, whereas black elements are required as context, i.e., they need to be present in the model in order to apply the rule. From here on, we omit the types of links to improve readability, as they can be inferred uniquely using the metamodels. The rule `ClassToDoc`, for instance, does not require any context and is therefore always applicable. It creates a class in the Java model and a document in the documentation model, and links the two elements via a correspondence. In contrast, the rule `SubClassToDoc` creates a subclass of an already existing class, along with a subtypes arrow. In the documentation model, a new document is created and referenced from the document that corresponds to the superclass in the Java model. A correspondence link is created between the new elements as well. The rules `MethodToEntry` and `FieldToEntry` are structurally similar to the rule `SubClassToDoc`: New methods and fields are added to a class (which can be a subclass in an inheritance hierarchy), while a corresponding entry is created in the document that corresponds to this class. For all four rules, so-called "attribute conditions"[1] are added in a textual syntax beneath the visual representation of the rule. These conditions ensure that the names of the linked elements (classes and documents, methods and entries, fields and entries) have the same name. The rule `AddParameter` does not create any element in the documentation model, but adds a parameter to a method in the Java model and links it to the entry of the method. There are also rules which only affect only the documentation model. `AddGlossary` creates a glossary node, `AddGlossaryEntry` adds new entries to it. Finally, the rule `LinkGlossaryEntry` connects document entries and glossary entries, which establishes the connection between the documents and the glossary.

The declarative rules define the language of a TGG, i.e., all triple graphs that can be generated with finitely many rule applications, starting from an empty triple. While this procedure is directly used to generate consistent models, the setting is different for other consistency management operations: For forward and backward transformation, one of the models is given as input (and may not be changed during the transformation process). For consistency checking, both source and target model are provided to the operation. Two variants are possible: Either the consistency checker tries to complete the input to a consistent triple by creating suitable correspondence links, denoted as correspondence creation (`CC`) in the following, or the correspondence model is provided as an additional input and only needs to be checked, which we refer to as check only (`CO`) in the following.

[1] While sophisticated definitions for attribute conditions in algebraic graph transformation [EEPT06] and TGGs [AVS12] have been presented in previous work, we restrict ourselves to only comparing attribute values locally. As this restricts the applicability of the TGG rule and can be directly attached to it, we do not handle attribute conditions separately.

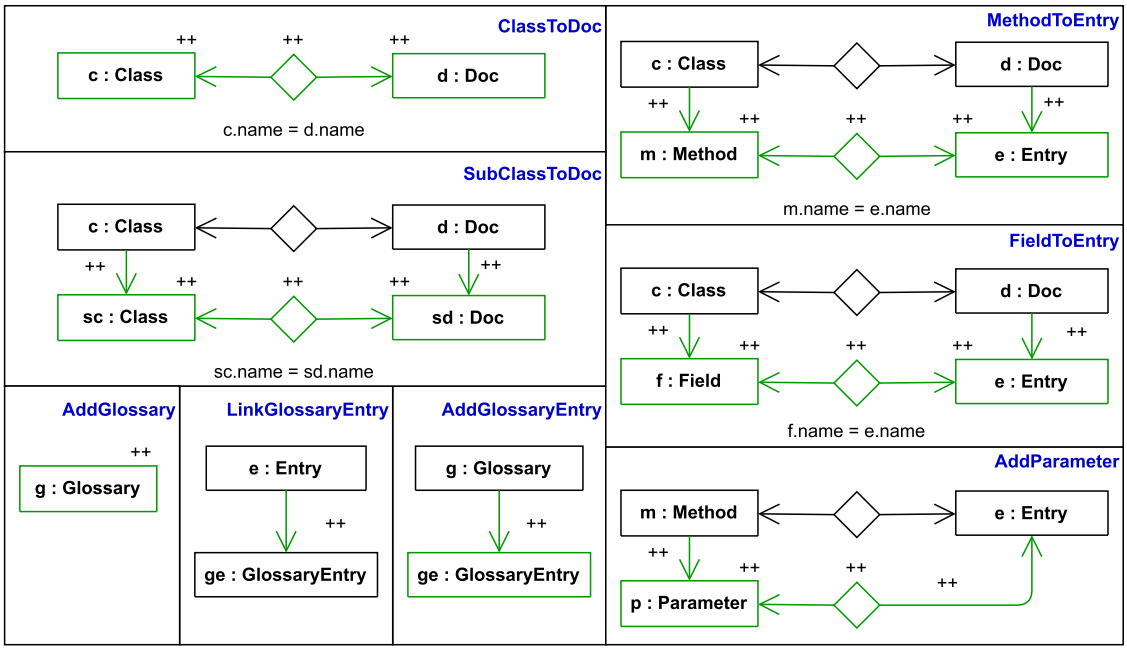


Fig. 3. TGG rules for `JavaToDoc`

Consistency management operations: The respective operational rules are derived from the declarative rules by requiring the green (created) elements of the given domain as additional context, and marking them as "translated" after applying the rule. Operational rules for `MethodToEntry` are depicted in Fig. 4. When performing a forward transformation (`FWD_OPT`) of a given Java model, the method `m` is required to exist before applying the rule, and marked as translated as indicated by the symbol $\square \rightarrow \boxtimes$. The suffix "OPT" stands for optimisation-based transformation, as opposed to conventional "greedy" strategies that successively (and irreversibly) apply rules at collected matches and therefore often lack completeness guarantees. As it must be guaranteed that the source model context is already translated (because it exists anyway as part of the input model), the class `c` is required to be marked already ($\boxtimes$). Conversely, the corresponding entry `e` is marked when performing a backward transformation (`BWD_OPT`), provided that the document `d` is already marked. The two consistency checkers mark elements of both Java and documentation model. The correspondence creation operation (`CC`), as the name suggests, creates only correspondences between marked elements, whereas the check only operation (`CO`) marks correspondence nodes as translated. For `FWD_OPT` and `BWD_OPT`, the attribute conditions are operationalised as well: As one of the attribute values if fixed by the additional context node that needs to be marked, the respective other value is assigned accordingly to respect the equality constraint.

Contribution: The novelty of the presented approach is the translation of the consistency management problem, i.e., the completion of the given input model(s) to a consistent triple that is both contained in the language of the TGG and satisfies all graph constraints, into an optimisation problem. In particular, we create an ILP that maximises the number of marked elements via a suitable objective function, with linear constraints guaranteeing language membership and graph constraint satisfaction. If it is possible to determine a solution that marks the input instance completely, we can conclude that a consistent transformation result was found, or the consistency check was successful, respectively. Otherwise, the operation yields a sub-triple with a maximum number of marked elements that is contained in the TGG's language and fulfils the specified constraints. Purely constraint-based approaches often suffer from severe scalability problems as they operate on the level of model elements and therefore involve numerous constraints that guarantee basic graph properties. Our approach, in contrast, addresses this problem by operating on rule application level, i.e., combining graph pattern matching and ILP in order to relate variables to entire matches and therefore significantly reducing the size of the constructed optimisation problem.

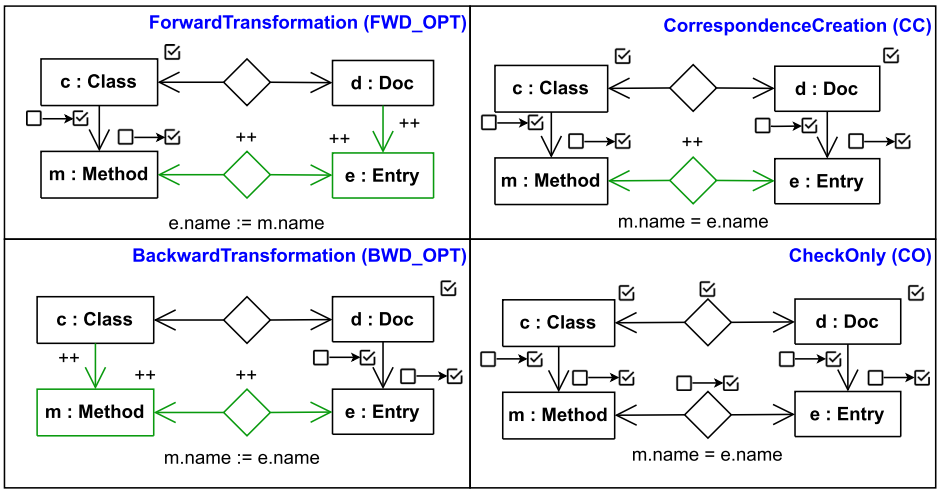


Fig. 4. Operational rules for `MethodToEntry`

Solution overview: To describe the solution process for a backward transformation and explain the construction of objective function and ILP constraints, we consider the example instance depicted in Fig. 5. In the documentation model, the nodes and edges are annotated in blue with the rule applications (also denoted as "direct derivations") d_i that potentially mark the respective element. The elements of the correspondence and Java model are depicted in grey (as they are not given as input but constructed during the transformation process) and represent a superset of all possible results of rule applications. Each element of the source and correspondence model is annotated with the rule application by which it is potentially created. In more detail, the rule applications (d_1) and (d_2) transform the respective documents into classes with the `ClassToDoc` rule. (d_3), in contrast, is an application of `SubClassToDoc` and marks the incoming edge of `doc2` as well. The entries `e4` and `e5` are transformed into methods by applying the rule `MethodToEntry` (d_4, d_5). The remaining rules only mark elements in the target model: While d_6 marks the glossary `g6` by applying `AddGlossary`, d_7 as an application of `AddGlossaryEntry` marks `ge7` and the connecting edge. Finally, d_8 and d_9 are applications of `LinkGlossaryEntry` and mark the edges between the entries and `ge7`. To determine the optimal solution, all rule applications d_i are associated with binary variables δ_i that are set to 1 if d_i is applied to form the solution graph (cf. Definition 5.6).

Furthermore, to encode the graph constraints into the optimisation problem, all possible matches for premises and conclusions are annotated to the involved elements. Negative constraints are hereby represented as graph constraints with a premise but no conclusions, as this is semantics-preserving: As soon as the premise (i.e. the negative constraint) can be matched, the constraint is violated. In the concrete example instance (Fig. 5), matches for premises (p_i) and conclusions (c_i) are annotated in red to the elements which they involve. The premises p_{10} and p_{11} refer to the `NoEmptyClass` constraint, and c_{13} and c_{14} form the respective conclusions for p_{10}. For p_{11}, no matches for conclusions are available, such that the constraint is violated as soon as the class `c2` is created. There is also one match each for the premise (p_{12}) and conclusion (c_{15}) of the `SameNameSameGlossaryEntry` constraint. The matches p_j and c_k for premises and conclusions are associated with binary variables π_j and γ_k (cf. Definition 5.7). In contrast to the binary variables for rule applications (d_i), these variables cannot be freely chosen and do not have any influence on the objective function. Their value assignment is rather dependent on the choice of the binary variables for rule applications, and restricts the set of valid solutions. The construction of the optimisation problem using these variables is presented in the following.

- **Context for rules:** Firstly, there must be ILP constraints that ensure that the application of a rule depends on the application of all rules that provide context for it. In the example instance, the application of `SubClassToDoc` (d_2) depends on the application of `ClassToDoc` as `c1`, `doc1` and their correspondence are required as context elements. Implication constraints of the form $di \Rightarrow (d_{j_1} \vee \cdots \vee d_{j_m}) \wedge \cdots \wedge (d_{k_1} \vee \cdots \vee d_{k_n})$ are thus created for all rule applications d_i with required context elements $j, \ldots, k$, and rule applications $(d_{j_1}, \ldots, d_{j_m}, \ldots, d_{k_1}, \ldots d_{k_n})$ that possibly mark these elements.

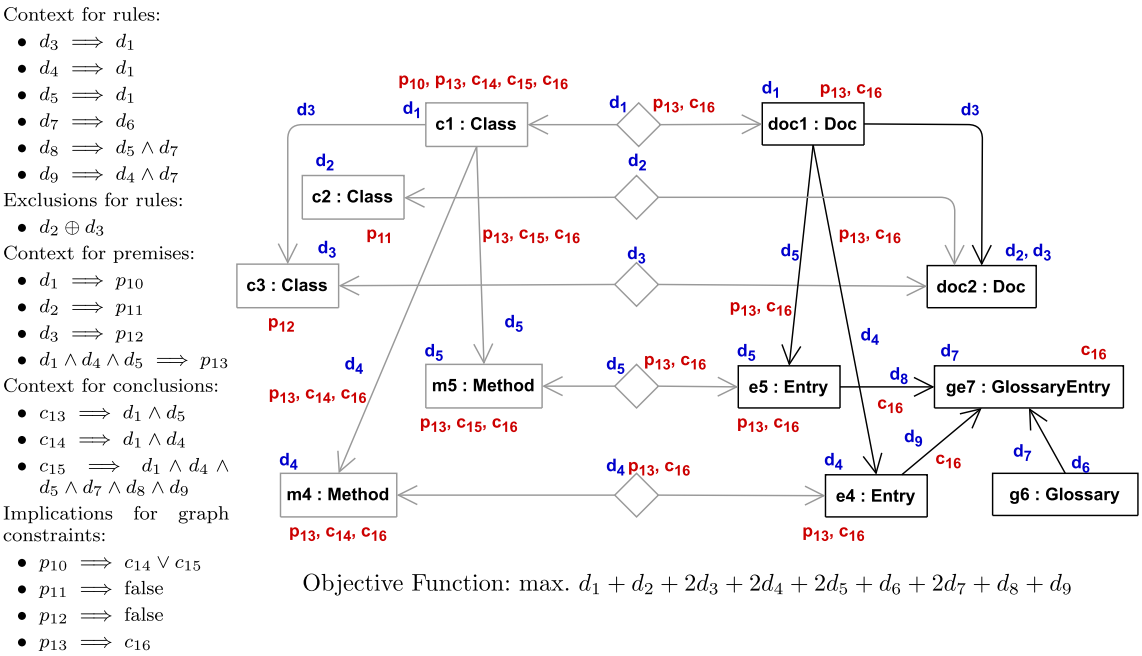


Fig. 5. Inconsistent example instance with annotations for rule applications and constraint matches

- **Exclusions for rules:** It must be prohibited that an element is marked more than once, because it would not be possible to create a single element multiple times with declarative rules. The only example here is the exclusion of d_2 and d_3 being applied both, as they mark the common element `doc2`. For each element that can be marked by multiple rule applications $d_i, \ldots, d_j$, an exclusion constraint $d_i \oplus \cdots \oplus d_j$ is created.

- **Context for premises:** Instead of influencing the objective function, graph constraints pose additional restrictions to the set of valid solutions, which means that they are translated into additional ILP constraints. Matches for premises depend on rule applications which mark or create the elements that are involved in the graph constraint. In this sense, the premise is fulfilled as soon as all context elements are marked (in the given part of the model) or created (in the remainder of the triple). However, as soon as the context is provided completely, the premise *is* fulfilled. The implication constraint is thus in the opposite direction: Choosing a subset of rule applications $d_i, \ldots, d_j$ that is sufficient to create the context for a premise match p_k implies that p_k is fulfilled. Taking a concrete example, the premise p_{13} for the graph constraint `SameNameSameGlossaryEntry` depends on the rule applications d_1, d_4 and d_5.

- **Context for conclusions:** Similar to premise constraints, it must also be reflected in the ILP under which conditions a conclusion of a graph pattern holds. In order to conclude a match for c_k, all involved nodes and edges must be marked or created by at least one rule application each. This subset $d_i, \ldots, d_j$ of rule applications is implied by c_k. In the concrete example, there are two matches (c_{13}, c_{14}) for the conclusion of `NoEmptyClass`, which require the rules d_1 and d_4, and $d1$ and $d5$, respectively, to be applied.

- **Implications for graph constraints:** The semantics of premise and conclusion(s) is reflected in the implications for graph constraints, which define that the presence of a premise match implies the existence of a corresponding conclusion match. For negative constraints, there are no conclusions, such that a solution for the ILP that fulfils this constraint cannot be valid. The class `c1` fulfils the constraint `NoEmptyClass` as long as one of the methods `m3` and `m4` are created, leading to an implication constraint $p_{10} \Rightarrow c_{14} \vee c_{15}$. The classes `c2` and `c3`, in contrast, do not have any fields or methods, such that the set of conclusions is empty. The respective implication constraints thus forbid the creation of these two classes.

- **Objective function:** As previously mentioned, the search for a consistent solution is driven by maximising the number of marked elements. If it is possible to mark the input models entirely, they can be completed to a triple that is contained in the TGG's language and fulfils all graph constraints. To form the objective function, a coefficient is computed for each binary variable d_i that reflects the number of elements that are potentially marked by the rule application. The sum of binary variables d_i weighted with these coefficients is the goal function to be maximised. Variables associated with graph constraints need not be taken into account because they do neither mark nor create elements.

All input model elements in the example instance can be marked setting d_1 and $d_3 \ldots d_9$ to 1 and d_2 to 0, leading to an objective function value of 12 equal to the total number of elements in the target model. This marking would however violate the constraint `NoEmptyClass` in the source model, as `c3` does not have any methods or fields. We will now analyse how this graph constraint violation becomes apparent in the ILP: As d_3 was chosen, it creates the class `c3`, on which the premise p_{12} of the constraint `NoEmptyClass` can be matched, and as a result, p_{12} must be set to 1 as well. This violates the constraint $p_{12} \Rightarrow$ false therefore the value assignment does not lead to a consistent transformation solution. The optimal solution, representing the maximal consistent sub-triple, is achieved by leaving out d_3, such that p_{12} can be set to 0 as well, resulting in an objective function value of 10. This means, however, that the document `doc2` and its incoming hyper-reference cannot be transformed unless another entry is added to it in the target model.

4. Preliminary definitions

This section briefly revises the fundamentals of algebraic graph transformation which are relevant for this article. Most definitions are adapted from Ehrig et al.[EEPT06], supplemented by the definition of schema compliance [AST12]. In TGGs, both the involved models and the consistency relation between them are represented in form of *graphs*. We therefore define graphs as objects and graph morphisms as arrows, which map nodes and edges of one graph to those of another.

Definition 4.1 (Graph (Morphism)).
A **graph** $G = (V, E, src, trg)$ consists of a set V of nodes (vertices), a set E of edges, and two functions $src, trg : E \to V$ that assign each edge a source and target node, respectively. The set $\text{elem}(G) = V \cup E$ denotes the union of vertices and edges, whereby it holds that $V \cap E = \emptyset$. Given graphs $G = (V, E, src, trg)$, $G' = (V', E', src', trg')$, a **graph morphism** $f : G \to G'$ consists of two functions $f_V : V \to V'$ and $f_E : E \to E'$ such that $src ; f_V = f_E ; src'$ and $trg ; f_V = f_E ; trg'$. The ; operator denotes the composition of functions: $f ; g(x) := g(f(x))$.

The definitions of graphs and graph arrows (Fig. 4.1) can be lifted in a straightforward manner to triple graphs and triple morphisms. Besides the interchangeable source and target models, the correspondence relation is represented in form of a graph as well, forming a *triple graph*. An example for such a triple graph (although not compliant with the TGG as stated in Sect. 3) is depicted in Fig. 5. Here, the Java model is denoted as source graph and the documentation model as the target graph, whereby this choice is just a question of design.

Definition 4.2 (Triple Graph (Morphism)).
A **triple graph** $G = G_S \overset{\gamma_S}{\leftarrow} G_C \overset{\gamma_T}{\rightarrow} G_T$ consists of graphs G_S, G_C, G_T and graph morphisms $\gamma_S : G_C \to G_S$ and $\gamma_T : G_C \to G_T$. *elem(G)* denotes the union $\text{elem}(G_S) \cup \text{elem}(G_C) \cup \text{elem}(G_T)$. A **triple morphism** $f : G \to G'$ with $G' = G'_S \overset{\gamma'_S}{\leftarrow} G'_C \overset{\gamma'_T}{\rightarrow} G'_T$, is a triple $f = (f_S, f_C, f_T)$ of graph morphisms where $f_X : G_X \to G'_X$, $X \in \{S, C, T\}$, $\gamma_S ; f_S = f_C ; \gamma'_S$ and $\gamma_T ; f_T = f_C ; \gamma'_T$.

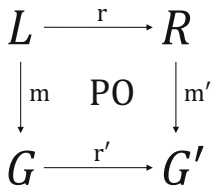
In this setting, we introduce typing by demanding a type (triple) morphism to a chosen type (triple) graph. In the rule `MethodToEntry` in Fig. 3, e.g., class nodes and method nodes can be distinguished by typing information. The language of a type (triple) graph TG is the set of (triple) graphs typed over TG.

Definition 4.3 (Typed Triple Graph (Morphism)).
A typed triple graph $(G, type)$ is a triple graph G together with a triple morphism $type : G \to TG$ to a distinguished type triple graph TG. A **typed triple morphism** $f : \hat{G} \to \hat{G}'$ is a triple morphism $f : G \to G'$ with $type = f ; type'$, where $\hat{G} = (G, type)$, $\hat{G}' = (G', type')$. $\mathcal{L}(TG) := \{G \mid \exists\ type : type(G) = TG\}$ denotes the set of all triple graphs of type TG.

In the following, all (triple) graphs and (triple) morphisms are assumed to be typed unless explicitly stated otherwise. Model transformations are expressed as applications of (triple) rules (cf. Fig. 3) , which are formalised as (triple) graph morphisms. The black elements of the rule form the left-hand side (L), whereas the union of black and green elements makes up the right-hand side (R). The morphism $r : L \to R$ specifies how green elements are added to the host graph G by applying the rule, such that a new (triple) graph is produced. Another requirement is a morphism m, denoted as *match*, that maps the left-hand side of the rule to the host graph. (Triple) rules are applied by constructing a *pushout*, which can be understood as a generalised union of (triple) graphs R and G over a common sub-(triple)graph L:

Definition 4.4 (Triple Rule (Application)).

A **triple rule** $r : L \to R$ is a monomorphic (injective) triple morphism. A direct derivation $G \overset{r@m}{\Longrightarrow} G'$ via a triple rule r, is constructed as depicted to the right by building a pushout over r and a triple monomorphism $m : L \to G$ called a match. A **derivation** $D : G \overset{*}{\Longrightarrow} G_n = G \overset{r_1@m_1}{\Longrightarrow} G_1 \overset{r_2@m_2}{\Longrightarrow} \cdots \overset{r_n@m_n}{\Longrightarrow} G_n$ is a sequence of direct derivations. We denote by $\mathcal{D} = \{d_1, \ldots, d_n\}$ the underlying set of direct derivations included in D.



A set of rules can be considered as a *grammar* that is sufficient to define a *language* of triple graphs. This language of a TGG can be produced by finitely many rule applications on the *empty triple graph*.

Definition 4.5 (Triple Graph Grammar (Language)).
A **triple graph grammar** TGG = $(TG, \mathcal{R})$ consists of a type triple graph TG, and a finite set $\mathcal{R}$ of triple rules. The **triple graph language** of TGG is defined as $\mathcal{L}(TGG) = \{G_\emptyset\} \cup \{G \mid \exists\ D : G_\emptyset \overset{*}{\Longrightarrow} G\}$, where $G_\emptyset$ is the **empty triple graph**.

It is clear at this point how the language of a TGG is formed via rule applications; We now define how to restrict this set of valid triples further via graph constraints. Graph constraints pose conditions on a graph that should hold independently of the TGG at hand. We shall handle graph constraints of the form $gc = P \to C_i$, which are either satisfied trivially, if there does not exist a match for the premise P, or if there exists at least one match for a conclusion C_i. Negative constraints are a special case of graph constraints for which the set of conclusions is empty. Positive constraints are of the form $G_\emptyset \to C$. If the distinction is necessary, we refer to the general form of graph constraints as implication constraints. For a more general treatment of graph constraints, the reader is referred to Habel et al. [HP09, EEHP06].

Definition 4.6 (Graph Constraint).
A **graph constraint** is a pair $gc = (p_\emptyset : G_\emptyset \to P, \{c_i : P \to C_i \mid i \in I\})$, for some index set I. P is referred to as the **premise** and $\{C_i \mid i \in I\}$ as the **conclusions** of the graph constraint gc. A triple graph G **satisfies** gc, denoted by $G \models gc$, iff $\forall\ m_p : P \to G, \exists\ i \in I\ \exists\ m_{c_i} : C_i \to G, [m_p = c_i;\ m_{c_i}]$, where $m_p, (m_{c_i})_{i \in I}$ are monomorphisms.

With the help of graph constraints, we can define *schema compliance* as a second part of our consistency requirement. A schema consists of a type triple graph TG and a set $\mathcal{GC}$ of graph constraints. In the running example, the triple of metamodels in Fig. 2 together with the constraints in Fig. 2 form such a schema. A (triple) graph *complies* to a schema if it is typed over TG and fulfils all graph constraints in $\mathcal{GC}$.

Definition 4.7 (Schema Compliance).
A schema is a pair $(TG, \mathcal{GC})$ of a type triple graph TG and a set $\mathcal{GC}$ of graph constraints. Let $\mathcal{L}(TG, \mathcal{GC}) := \{G \in \mathcal{L}(TG) \mid \forall\, gc \in \mathcal{GC},\ G \models gc\}$ denote the set of all **schema-compliant** triple graphs.

5. Formalisation

Based on the fundamental definitions of Sect. 4, we now formalise further concepts to be able to show *correctness* and *completeness* properties in Sect. 6, i.e., that consistency management operations terminate with a consistent result if and only if it exists. The concepts and proofs combine ideas of seminal work on consistency management with TGGs and ILP, so parts of the formalisation are adapted from these sources. Firstly, Leblebici et al. [LAS17, Leb18] presented a correspondence creation approach (CC) and showed that correctness and completeness can be

guaranteed posing a few restrictions on the TGG. The formal concepts and proofs were extended by Weidmann et al. towards other consistency management operations on the one hand [WALS19], and graph constraints for check only operations (CO) on the other hand [WA20]. An overview of the involved publications and their novel contribution is provided in Fig. 5, whereby ∗ denotes this article. In the following, we extend the hybrid approach based on TGGs and ILP to be applicable in a setting with various consistency management operations. In particular, the definition of consistent input and solution is extended to graph constraints (Definition 5.2), we define how created and marked elements provide context for premise and conclusion patterns (Definition 5.5) and, based on this redefinition, the ILP constraints for guaranteeing context are adapted in Definition 5.10.

Via TGGs, a language of (consistent) triples is defined. It is composed of all graphs that can be *generated* by applying triple rules, creating elements in each domain (source, correspondence, and target) simultaneously. However, this definition is not sufficient to specify e.g. forward and backward transformations because parts of the triple are already given, such as the target model in Fig. 5. These parts differ for all operations and can be combinations of the source, target and correspondence graphs, which we denote as the *starting triple graph*.

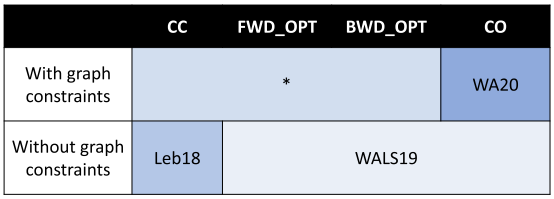


Fig. 6. Overview of publications

Definition 5.1 (Starting Triple Graph).
For each operation $op \in \{\text{CC, CO, FWD\_OPT, BWD\_OPT}\}$, a *starting triple graph* G_0 for a triple graph G is defined as:

Operation	Starting Triple Graph (G_0)
CC	$G_S \leftarrow \emptyset \rightarrow G_T$
CO	$G_S \leftarrow G_C \rightarrow G_T = G$
FWD_OPT	$G_S \leftarrow \emptyset \rightarrow \emptyset$
BWD_OPT	$\emptyset \leftarrow \emptyset \rightarrow G_T$

The starting triple graph of an operation is a *consistent input* if it can be extended to a *consistent solution* by applying the rules of the operation, i.e., a triple graph that is contained in the language of the TGG and is compliant to the given schema. The documentation model of Fig. 5 is *not* a consistent input in this sense, as all possible backward transformations result in triples that violate at least one of the constraints.

Definition 5.2 (Consistent Input and Consistent Solution).
Given a triple graph grammar TGG and a schema $(TG, \mathcal{GC})$, a starting triple graph $G_0 = G_S \leftarrow G_C \rightarrow G_T$ is said to be *consistent input* iff $\exists\ G' = G'_S \leftarrow G'_C \rightarrow G'_T \in \mathcal{L}(TGG) \cap \mathcal{L}(TG, \mathcal{GC})$, such that:

Operation	Conditions
CC	$G_S = G'_S, G_T = G'_T$
CO	$G_S = G'_S, G_C = G'_C, G_T = G'_T$
FWD_OPT	$G_S = G'_S$
BWD_OPT	$G_T = G'_T$

G' is referred to as a *consistent solution* for G_0 in each case.

To determine whether a triple graph is a member of the language of a TGG, one tries to produce this triple graph by forming a derivation sequence starting with the empty triple graph (cf. Definition 4.5). However, the different consistency management operations start with their starting triple graph instead. The declarative rules must therefore be transformed to an *operationalised* form, meaning that the green (created) elements of the declarative TGG rule are either marked (in the given input) or created (in the remainder of the triple). The elements of the starting triple graph are denoted as *markable elements*, and the elements to be created by the operation as *created elements*. In the example instance of Fig. 5, the markable elements are black, whereas the created elements are grey.

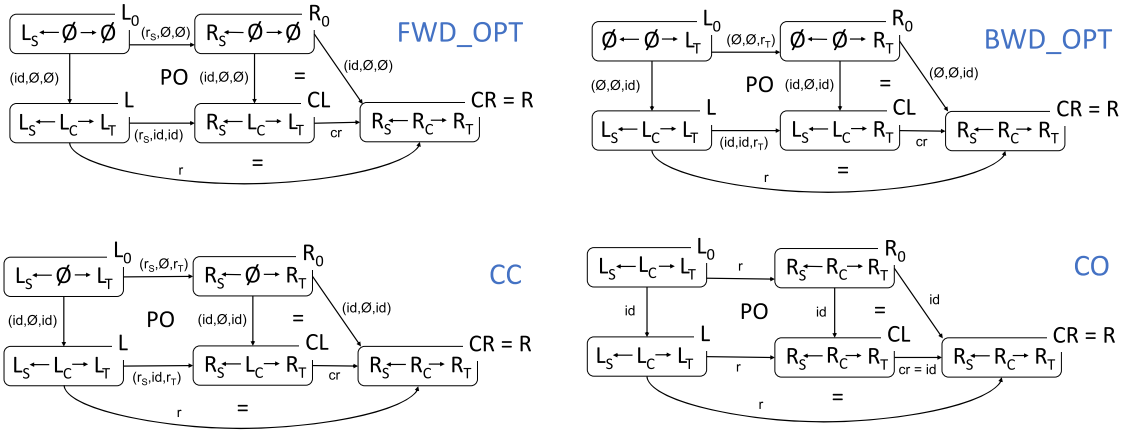


Fig. 7. Operational rules for FWD_OPT, BWD_OPT, CC and CO

Definition 5.3 (Markable Elements and Created Elements).
The sets *mrkElem*(G) and *crtElem*(G) are defined for a triple graph $G = G_S \leftarrow G_C \rightarrow G_T$ as follows:

Operation	mrkElem(G)	crtElem(G)
CC	elem(G_S) ∪ elem(G_T)	elem(G_C)
CO	elem(G)	∅
FWD_OPT	elem(G_S)	elem(G_C) ∪ elem(G_T)
BWD_OPT	elem(G_T)	elem(G_S) ∪ elem(G_C)

Consequently, elem(G) = mrkElem(G) ∪ crtElem(G).

To formalise the rules that partly mark and create elements, we define them as *operational* rules derived from declarative rules (Definition 5.4). Depending on the operation, the left-hand side CL of the operational rule can be formed out of the left-hand side (L) of the declarative rule, and the starting triple graph for the right-hand side (R_0) via a pushout construction. The right-hand side CR of the operational rule is equal to the right-hand side of the original rule. The elements that are added by the morphism $cl : L \rightarrow CL$ are denoted as *marking elements* of the operational rule cr. In Fig. 4, operational rules for the `MethodToEntry` rule are depicted for all four operations.

Definition 5.4 (Operational Rule and Marking Elements).
Given a triple rule $r : L \rightarrow R$, let L_0 and R_0 be the starting triple graphs for L and R. The **operational rule** $cr : CL \rightarrow CR$ for r is constructed as depicted in Fig. 7. CL is formed via pushout construction of L and R_0 over L_0. It holds that $CR = R$, and $cr : CL \rightarrow CR$ is induced via the universal property of the pushout. An element $e \in$ mrkElem(CL) is a **marking element** of cr iff $\nexists e' \in$ mrkElem(L) with $r_S(e') = e$ or $r_C(e') = e$ or $r_T(e') = e$.

Considering the elements of the host graph that are involved in an operational rule application, a partition into four sets is possible: On the one hand, elements can be *marked* or *created* by the rule application, depending on whether they are contained in the starting triple graph. On the other hand, some elements are *required* as context in both the operational and the original TGG rule. If such a context element is also part of the starting triple graph, it must also be marked already, simulating the behaviour of the generative TGG specification. In Fig. 5, elements which are annotated with $\square \rightarrow \boxtimes$ are marked by the rule, whereas the ☑ annotation indicates that they must be marked already. As for declarative rules, created elements have a ++ mark-up and are coloured green, and context elements are black and do not have any mark-up.

As graph constraints do not mark elements, only sets for the elements that are required to be marked or created already in order to form premise and conclusion, respectively, are defined. It is assumed that the names of elements in G and G' are preserved by the rule application cr, such that they can be identified with each other.

Definition 5.5 (Marked, Created and Required Elements).
For a direct derivation $d : G \overset{cr@cm}{\Longrightarrow} G'$ via an operational rule $cr : CL \to CR$, the following sets are defined:

- crt(d) = crtElem(G') \ crtElem(G)
- mrk(d) = {e ∈ elem(G) | ∃ $e' \in$ elem(CL), $cm(e') = e$ where e' is a marking element of cr}
- reqMrk(d) = {e ∈ mrkElem(G) | ∃ $e' \in$ elem(CL), $cm(e') = e$ where e' is not a marking element of cr}
- reqCrt(d) = {e ∈ crtElem(G) | ∃ $e' \in$ elem(CL), $cm(e') = e$}

For a graph constraint $gc = (P, \{c_i : P \to C_i \mid i \in I\})$ and morphisms $m_p : P \to G, (m_{c_i})_{i \in I} : C_i \to G$, we define:

- reqMrk(m_p) = {$e \in$ mrkElem(G) | ∃ $e' \in$ elem(P), $m_p(e') = e$}
- reqCrt(m_p) = {$e \in$ crtElem(G) | ∃ $e' \in$ elem(P), $m_p(e') = e$}
- reqMrk(m_{c_i}) = {$e \in$ mrkElem(G) | ∃ $e' \in$ elem(C_i), $m_{c_i}(e') = e$}, $i \in I$
- reqCrt(m_{c_i}) = {$e \in$ crtElem(G) | ∃ $e' \in$ elem(C_i), $m_{c_i}(e') = e$}, $i \in I$

In general, there are more rule application candidates that can be found by matching the operational rules than are necessary to form a derivation sequence from the starting triple graph to a consistent solution. The subset of necessary rule applications is therefore determined by transforming the graph problem into a search problem to be solved by e.g. an ILP solver, in which each rule application candidate is associated to a binary variable. Its value in the retrieved solution (0 or 1) indicates whether the candidate is considered for forming the final derivation sequence. In Fig. 5, these constraints are shown on the top left ("context for rules", "exclusion for rules") in the notation of propositional logic, as this is more intuitive for a first understanding.

Definition 5.6 (Constraints for Derivations).
Given a starting triple graph G_0, let $D : G_0 \overset{*}{\Longrightarrow} G_n$ be a derivation via operational rules with the underlying set $\mathcal{D}$ of direct derivations. For each direct derivation $d_1, \ldots, d_n \in \mathcal{D}$, respective binary variables $\delta_1, \ldots, \delta_n \in \{0, 1\}$ are defined. A linear constraint $\mathcal{LC}$ for $\mathcal{D}$ is a conjunction of linear inequalities which involve $\delta_1, \ldots, \delta_n$. A set $\mathcal{D}' \subset \mathcal{D}$ fulfils $\mathcal{LC}$, denoted as $\mathcal{D}' \vdash \mathcal{LC}$, iff $\mathcal{LC}$ is satisfied for variable assignments $\delta_i = 1$ if $d_i \in \mathcal{D}'$ and $\delta_i = 0$ if $d_i \notin \mathcal{D}'$, $1 \leq i \leq n$.

Similar to rule applications, matches for graph constraints are also associated to binary variables to ensure that the retrieved solution is schema compliant, i.e., respects all specified constraints. Thereby, matches for premises and conclusions are separately encoded into constraints as they depend on different sets of rule applications. The interdependencies between premises and conclusions are expressed by further constraints as well. In contrast to the binary variables for rule applications, the value assignment cannot be chosen by the ILP solver. The respective linear constraints, which encode interdependencies between rule applications and graph constraints, are formulated in a way that any variable assignment that does not violate them leads to a schema compliant solution. For the running example, the constraints for graph constraints are listed on the bottom left of Fig. 5 ("context for premises", "context for conclusions", "implications for graph constraints").

Definition 5.7 (Constraints for Graph Constraints).
Let $\mathcal{GC} = \{(P, \{c_i : P \to C_i \mid i \in I\})\}$ be a set of graph constraints. Let $\mathcal{P} = \bigcup_{gc \in \mathcal{GC}} \{m_p : P \to G\}$ be a set of premises and $\mathcal{C} = \bigcup_{gc \in \mathcal{GC}} \{m_{c_i} : C_i \to G, i \in I\}$ a set of conclusions. For each premise $m_p \in \mathcal{P}$, respective binary variables $\pi_1 \ldots \pi_n$, and for each conclusion $m_{c_i} \in \mathcal{C}$, respective binary variables $\gamma_{1,1} \ldots \gamma_{1,m_1} \ldots \gamma_{n,1} \ldots \gamma_{n,m_n}$ are defined. A linear constraint $\mathcal{LC}$ for $\mathcal{GC}$ is a conjunction of linear inequalities which involve $\pi_1 \ldots \pi_n$ and $\gamma_{1,1} \ldots \gamma_{1,m_1} \ldots \gamma_{n,1} \ldots \gamma_{n,m_n}$. A triple graph G fulfils $\mathcal{LC}$, denoted as $G \vdash \mathcal{LC}$, iff $\mathcal{LC}$ is satisfied for any variable assignment $\{\pi_1 \ldots \pi_n\} \to \{0, 1\}$, $\{\gamma_{1,1} \ldots \gamma_{1,m_1} \ldots \gamma_{n,1} \ldots \gamma_{n,m_n}\} \to \{0, 1\}$.

In the following, we will describe how to construct constraints that ensure that the chosen rule applications form a valid derivation sequence from the starting triple graph to the solution.

As markings in operational rules correspond to the creation of elements in the original rules of the underlying TGG, it must be prohibited that elements are marked multiple times as this would mean that an element is created

more than once. For each node and edge, a predicate *mrkSum* of type integer is defined that reflects the number of markings per element by counting the rule applications that mark this element. In Fig. 5, mrkSum would be 2 for the document `doc2`, and 1 for all other elements of the target model.

Definition 5.8 (Sum of Alternative Markings for an Element).
Given a starting triple graph G_0, let $D : G_0 \overset{*}{\Longrightarrow} G_n$ be a derivation via operational rules with the underlying set $\mathcal{D}$ of direct derivations. For each element $e \in \text{elem}(G_0)$, let $\mathcal{E}(e) = \{d \in \mathcal{D} \mid e \in \text{mrk}(d)\}$. The integer $\text{mrkSum}(e)$ denotes the sum of the associated variable assignments for each $d \in \mathcal{E}$:

$$\text{mrkSum}(e) = \sum_{d_i \in \mathcal{E}(e)} \delta_i$$

This is used in a linear constraint to ensure that elements are marked at most once. In Fig. 5, this is denoted as "exclusions for rules", whereby all non-trivial constraints are omitted for clarity.

Definition 5.9 (Constraint 1: Mark Elements at Most Once).
Given a starting triple graph G_0, let $D : G_0 \overset{*}{\Longrightarrow} G_n$ be a derivation via operational rules:

$$\text{markedAtMostOnce}(G_0) = \bigwedge_{e \in \text{elem}(G_0)} [\,\text{mrkSum}(e) \leq 1]$$

With respect to this definition, the constraint is also fulfilled if there are elements that are not marked at all, resulting in a $mrkSum$ of 0. Consequently, there are valid solutions that cannot mark the input entirely. The reason for the sum of marked elements not being strictly equal to 1 is the desired treatment of inconsistent input: In this case, a consistent sub-triple is returned by the ILP solver.

Another constraint must ensure that an operational rule is applicable if and only if the respective declarative rule would be applicable (in a setting in which the same derivation sequence is followed). This means that all context elements of the starting triple graph must be marked, and context elements of the remaining part of the triple must be created already. Similarly, as soon as all rule applications are chosen that create and mark the respective contexts, the binary variables associated to premises and conclusions must be assigned accordingly. In total, this guarantees that the respective TGG rule is applicable in the corresponding situation, and that the marked and created parts of the graph are schema compliant up to that point. The constraints which form the context predicate are listed on the left of Fig. 5), ordered by whether they refer to rule applications, premises or conclusions.

Definition 5.10 (Constraint 2: Guarantee Context).
Given a starting triple graph G_0 and a schema $(TG, \mathcal{GC})$, let $D : G_0 \overset{*}{\Longrightarrow} G_n$ be a derivation via operational rules with the underlying set $\mathcal{D}$ of direct derivations. For each direct derivation $d \in \mathcal{D}$, each morphism $m_p \in \mathcal{P}$ and each morphism $m_{c_i} \in \mathcal{C}$, the following constraints are defined:

$$\text{context}(d) = \bigwedge_{e \in \text{req}(d)} [\delta \leq \text{mrkSum}(e)] \wedge \bigwedge_{d_j \in \mathcal{D}, [\text{reqCrt}(d) \cap \text{crt}(d_j) \neq \emptyset]} [\delta \leq \delta_j]$$

$$\text{context}(m_p) = \sum_{e \in \text{reqMrk}(m_p)} [\text{mrkSum}(e) - 1] + \sum_{d_j \in \mathcal{D}, [\text{reqCrt}(m_p) \cap \text{crt}(d_j) \neq \emptyset]} [\delta_j - 1] \leq \pi - 1$$

$$\text{context}(m_{c_i}) = \bigwedge_{e \in \text{reqMrk}(m_{c_i})} [\gamma_i \leq \text{mrkSum}(e)] \wedge \bigwedge_{d_j \in \mathcal{D}, [\text{reqCrt}(m_{c_i}) \cap \text{crt}(d_j) \neq \emptyset]} [\gamma_i \leq \delta_j],\ i \in I$$

$$\text{context}(D) = \bigwedge_{d \in \mathcal{D}} \text{context}(d)$$

$$\text{context}(G) = \bigwedge_{m_p \in \mathcal{P}} context(m_p) \wedge \bigwedge_{m_{c_i} \in \mathcal{C}} context(m_{c_i})$$

For an intuitive understanding of these constraints, it is best to assume that the constraint markedAtMostOnce (Definition 5.9) already holds, i.e., elements are marked by at most one derivation. For created elements, note that the transformation process guarantees inherently that every created element is created by only one derivation (see Definition 4.4). $\text{context}(d)$ ensures that all marked elements required by d are indeed present (first part), and that all created elements required by d are created in the final result (second part). $\text{context}(m_p)$ is an implication of the form "m_p matches" $\Rightarrow \pi$. The two summands in the left part of the inequality are both 0 exactly when all required marked and created elements for m_p are present and are both negative otherwise. Demanding their sum

to be $\leq \pi - 1$ forces the solver to set π to 1 whenever m_p matches. context(c_i) is analogous to context(d), i.e., the solver is not allowed to set any γ_i to 1 unless m_{c_i} matches. According to this constraint the solver has no reason to set any γ_i to 1; this will only be enforced with Definition 5.11 that covers the relation between m_ps and m_{c_i}s. This last constraint type encodes the semantics of premise and conclusions of graph constraints (cf. Definition 4.6), such that schema compliance can be guaranteed for the transformation result. As the binary variables π (premise) and γ_i (conclusion) are set to 1 if the respective context is created or marked entirely, the potential matches can be considered as actual matches. The constraint can be formulated independent of the concrete rule applications, as their values influence the value assignment to all variables π and γ_i. For the running example, the respective constraints are shown on the bottom left ("implications for graph constraints").

Definition 5.11 (Constraint 3: Satisfy Graph Constraints).
Let $(TG, \mathcal{GC} = \{(P, \{c_i : P \to C_i \mid i \in I\})\})$ be a schema. A linear constraint $sat(G)$ expressing that G fulfils all graph constraints of $\mathcal{GC}$ is defined as follows:

$$sat(G) = \bigwedge_{m_p \in \mathcal{P}} [\pi \leq \sum_{m_{c_i} \in \mathcal{C}, m_p = c_i;\ m_{c_i}, i \in I} \gamma_i]$$

Furthermore, there are constellations in which rule application candidates mutually provide context for each other by marking or creating elements that are necessary to apply the respective other rule. In this manner, a *dependency cycle* is formed, such that none of the involved rules can ever be applied first due to missing context elements. There is no example for such a dependency cycle in the running example, though. To express a cyclic dependency, a relation $\triangleright$ among rule applications is introduced. Each subset $\{d_1, \ldots, d_n\}$ of rule applications that does not contain a cycle can be sequenced over this relation in a proper order.

Definition 5.12 (Dependency Cycles).
Let $D : G_0 \overset{*}{\Longrightarrow} G_n$ be a derivation via operational rules with the underlying set $\mathcal{D}$ of direct derivations. Relations $\triangleright, \triangleright^M, \triangleright^C \subseteq \mathcal{D} \times \mathcal{D}$ between $d_i, d_j \in \mathcal{D}$ are defined as follows:

$d_i \triangleright^M d_j$ iff $\text{reqMrk}(d_i) \cap \text{mrk}(d_j) \neq \emptyset$
$d_i \triangleright^C d_j$ iff $\text{reqCrt}(d_i) \cap \text{crt}(d_j) \neq \emptyset$
$d_i \triangleright d_j$ iff $(d_i \triangleright^M d_j) \vee (d_i \triangleright^C d_j)$

A set $cy \subseteq \mathcal{D}$ with $cy = \{d_1, \ldots, d_n\}$ of direct derivations is a **dependency cycle** iff $d_1 \triangleright \cdots \triangleright d_n \triangleright d_1$.

The first relation ($\triangleright^M$) holds if d_i requires an element to be marked and d_j marks it. Likewise, the second relation ($\triangleright^C$) holds if d_i requires an element to be created and d_j creates it. Combined with the other constraints that ensure that elements are only created and marked by a single direct derivation, these two relations can be combined to express that d_i depends on d_j. Cycles in this combined relation must, therefore, be prohibited. The following constraint forbids choosing all rule applications involved in a dependency cycle for the final solution, such that it is always possible to arrange the chosen rule applications properly.

Definition 5.13 (Constraint 4: Forbid Dependency Cycles).
Given a starting triple graph G_0, let $D : G_0 \overset{*}{\Longrightarrow} G_n$ be a derivation via operational rules with the underlying set $\mathcal{D}$ of direct derivations, and let $\mathcal{CY}$ be the set of all dependency cycles $cy \in \mathcal{D}$. A linear constraint acyclic(D) is defined as follows:

$$\text{acyclic}(D) = \bigwedge_{cy \in \mathcal{CY}, cy = \{d_1, \ldots, d_n\}} \sum_{i=1}^{n} \delta_i < n$$

Finally, we define the objective function to maximize the number of markings (bottom line of Fig. 5)), such that a solution that entirely marks the input (and can therefore be completed to a triple contained in the TGG's language) is preferred over a solution with less markings. As the solution must fulfil all ILP constraints, correctness constraints and schema compliance according to Definition 4.7 can be guaranteed.

Definition 5.14 (Optimisation Problem).
Given a starting triple graph G_0, a TGG (G, $\mathcal{R}$) and a schema $(TG, \mathcal{GC})$, let $D : G_0 \overset{*}{\Longrightarrow} G_n$ be a derivation via operational rules. The ILP to be optimised is constructed as follows:

$$\text{max.} \sum_{d \in D} |mrk(d)| \text{ s.t. markedAtMostOnce}(G_0) \wedge \text{context}(D) \wedge \text{context}(G_n) \wedge \text{acyclic}(D) \wedge \text{sat}(G_n)$$

6. Correctness and completeness

In the following, we show that the ILP-based solution strategy - posing some assumptions on the TGG in use - always terminates, and yields a consistent solution with respect to Definition 5.2 iff such a solution exists. Similar to the formalisation (Sect. 5), arguments from previous work [Leb18, WALS19, WA20] are combined and extended. The main challenge here is to show that schema-compliance can still be guaranteed when further elements are continuously added during match collection. In the following, let $TGG = (G_\emptyset, \mathcal{R})$ and a schema $(TG, \mathcal{GC})$ be given for all definitions, lemmas and theorems.

As previously stated, the goal of the optimisation step is to determine a subset of rule applications that forms a derivation sequence from the starting triple graph to a transformation result. We define a *proper subset* of operational rule applications that can be arranged such that a (possibly incomplete) derivation sequence is formed. The values assigned to the associated binary variables δ_i for each $d_i \in \mathcal{D}' \subseteq \mathcal{D}$ form a feasible solution for the ILP, i.e., satisfy all defined constraints (Definition 5.9, 5.10, 5.13 and 5.11).

Definition 6.1 (Proper Subset of Rule Applications).
Given a starting triple graph G_0, let $D : G_0 \overset{*}{\Longrightarrow} G_n$ be a derivation via operational rules with underlying set of direct derivations $\mathcal{D}$, and let $\mathcal{D}' \subseteq \mathcal{D}$ be a subset of direct derivations, such that $D' : G_0 \overset{*}{\Longrightarrow} G'$.

We refer to $\mathcal{D}'$ as a proper subset of $\mathcal{D}$ iff $\mathcal{D}' \vdash \text{markedAtMostOnce}(G_0) \wedge \text{context}(D') \wedge \text{context}(G') \wedge \text{acyclic}(D') \wedge \text{sat}(G')$.

The first lemma (Lemma 6.1) states that such a proper subset exists if and only if there is a triple graph G' that is contained in the TGG's language, fulfils all constraints and does not have more elements than the starting triple graph in the given parts of the triple.

Lemma 6.1 (Consistent Portions of a Triple Graph).
Given a starting triple graph G_0, let $D : G_0 \overset{*}{\Longrightarrow} G_n$ be a derivation via operational rules with underlying set of direct derivations $\mathcal{D}$.

$\exists$ proper subset $\mathcal{D}' \subseteq \mathcal{D}$ with $D' : G_0 \overset{*}{\Longrightarrow} G' \iff \exists\ G' \in \mathcal{L}(TGG) \cap \mathcal{L}(TG, \mathcal{GC})$ such that:

$$\text{mrkElem}(G') \subseteq \text{mrkElem}(G_0)$$
$$\text{mrkElem}(G') = \bigcup_{d' \in \mathcal{D}'} \text{mrk}(d'),$$
$$\text{crtElem}(G') = \bigcup_{d' \in \mathcal{D}'} \text{crt}(d').$$

Proof. (Sketch) The direct derivations $d \in \mathcal{D}'$ of a proper subset can be sequenced over the $\triangleright$ relation according to Def. 5.12, which is equivalent to the existence of a derivation D' which forms a triple graph G' by being applied on the starting triple graph G_0. G' consists of all elements which are marked or created by any $d \in \mathcal{D}'$. $G' \in \mathcal{L}(TGG) \cap \mathcal{L}(TG, \mathcal{GC})$ holds by Definition 6.1, guaranteeing language membership and schema compliance. □

The sequential application of rules of a proper subset leads to a transformation result, of which the marked and created elements form a triple that is member of the TGG's language and that is schema-compliant. In general, however, elements of the input can remain unmarked, which means that they were not (yet) consistently transformed. Therefore, we denote proper subsets as *maximal* which involve a maximal number of markings. As a proper subset fulfils all ILP constraints by definition, and maximising the number of marked elements is the optimisation goal, a maximal proper subset will be returned by the ILP solver.

Definition 6.2 (Maximal Proper Subset of Rule Applications).
Given a starting triple graph G_0, let $D : G_0 \overset{*}{\Longrightarrow} G_n$ be a derivation via operational rules with underlying set of direct derivations $\mathcal{D}$. A proper subset $\mathcal{D}'$ of $\mathcal{D}$ is *maximal* if there does not exist any other proper subset $\mathcal{D}''$ of $\mathcal{D}$ with a greater objective function value (cf. Definition 5.14).

Accordingly, we denote the triple graph that results from sequentially applying rules of the maximal proper subset on the starting triple graph as *maximally marked*.

Definition 6.3 (Maximally Marked Triple Graph).
Given a starting triple graph G_0, let $D : G_0 \overset{*}{\Longrightarrow} G_n$ be a derivation via operational rules with underlying set of

direct derivations $\mathcal{D}$. Let $\mathcal{D}'$ be a maximal, proper subset of $\mathcal{D}$. The triple graph G' identified with $\mathcal{D}'$ according to Lemma 6.1 is denoted as a *maximally marked triple graph* with respect to D.

In Theorem 6.1, the correctness of the ILP-based transformation can be shown using the notion of a maximally marked triple graph, i.e., if it is possible to mark each element of the starting triple graph G_0 exactly once, the resulting maximally marked triple graph is a consistent solution according to Definition 5.2.

Theorem 6.1 (Correctness).
Given a starting triple graph G_0 and a derivation $D : G_0 \overset{*}{\Longrightarrow} G_n$ for $op \in \{\text{CC, CO, FWD\_OPT, BWD\_OPT}\}$, and let $\mathcal{D}' \subseteq \mathcal{D}$ be a maximal proper subset of direct derivations, such that $D' : G_0 \overset{*}{\Longrightarrow} G'$. It holds for a maximally marked triple graph G' with respect to $\mathcal{D}$:

$$\bigcup_{d \in \mathcal{D}'} mrk(d) = \text{elem}(G_0) \Rightarrow G' \text{ is a consistent solution}$$

Proof. (Sketch) For G' being a consistent solution, (1) $G' \in \mathcal{L}(TGG)$ und (2) $G' \in \mathcal{L}(TG, \mathcal{GC})$ are required. The premise of the theorem, i.e., the marked part of G' and the non-empty part of G_0 are identical, states that it is possible to entirely mark G_0 by sequentially applying all direct derivations $d \in \mathcal{D}'$, whereby D' of a maximal, proper subset. By applying Lemma 6.1 with a maximal, proper subset, a triple graph G' can be produced that fulfils both conditions. Therefore, G' is a consistent solution according to Definition 5.2, while G_0 is a consistent input. □

For showing completeness, it remains to show the opposite direction, i.e., that the proposed strategy finds a consistent solution if one exists. The main challenge is to guarantee that the set $\mathcal{D}$ of rule application candidates is finite, such that the process always terminates, because ILP solving is known to be correct and complete as well. Therefore, we have to demand that the underlying TGG be *progressive* (cp. [GHL14] for bookkeeping mechanisms for TGGs), which means that each operational rule has to mark at least one element. It is possible that some TGGs are progressive for only a few operations. In the running example (cf. Fig. 3), progressiveness is not fulfilled for the forward transformation operation, as the rules `AddGlossary`, `LinkGlossaryEntry` and `AddGlossaryEntry` only operate on the target model. Likewise, as the rule `AddParameter` does not modify the target model, the TGG is not progressive for the backward direction either. For practical implementations, the problem can be overcome by applying heuristics, such as fixing an upper bound for applications of such rules.

Definition 6.4 (Progressive TGGs).
A TGG is progressive for an operation $op \in \{\text{CC, CO, FWD\_OPT, BWD\_OPT}\}$ if each of its operational rules for op has at least one marking element.

Although progressiveness of a TGG ensures that the number of markings strictly increases when following a derivation sequence, it is still possible that new rule applications create new context elements which themselves enable further rule applications, such that termination cannot be immediately guaranteed as elements of the starting triple graph can be marked infinitely often (although only one of these markings can be finally chosen). However, as soon as rule applications overlap in their markings and one of them depends on the created context of another, the dependent rule application is superfluous as it implies and excludes the application of another rule at the same time. Therefore, only *essential* rule applications (cf. Definition 6.5) should be considered for forming a derivation sequence, meaning that (1) identical rule applications and (2) rule applications that are in conflict with their create dependencies as described above (2) must be discarded. Note that in contrast to dependencies via marked context ($\triangleright^M$), create dependencies via $\triangleright^C$ are *actual* and not potential dependencies, as the rule application that creates an element is unique.

Definition 6.5 (Essential and Superfluous Rule Applications).
Given a starting triple graph G_0 and a derivation $D : G_0 \overset{*}{\Longrightarrow} G_n$ with underlying set of direct derivations $\mathcal{D}$. Let $\triangleright^C_* \subseteq \mathcal{D} \times \mathcal{D}$ be the transitive closure of the $\triangleright^C$ relation in Definition 5.12. A rule application $d_{n+1} : G_n \overset{cr_{n+1}@cm_{n+1}}{\Longrightarrow} G_{n+1}$ with operational rule cr_{n+1} is *essential* for D if:

1. $\nexists d_i \in \mathcal{D}, d_i : G_{i-1} \overset{cr_i@cm_i}{\Longrightarrow} G_i$ such that $cr_{n+1} = cr_i$ and $cm_{n+1} = cm_i$ and
2. $\text{mrk}(d_{n+1}) \cap \bigcup_{d' \in \mathcal{D}, d_{n+1} \triangleright^C_* d'} \text{mrk}(d') = \emptyset$.

Otherwise, d_{n+1} is *superfluous* for D.

Introducing some terminology for derivation sequences that purely consist of essential rule applications, such derivations are denoted as *final* according to Definition 6.6.

Definition 6.6 (Final Derivations with Operational Rules).
Given a starting triple graph G_0, let $D : G_0 \overset{*}{\Longrightarrow} G_n$ be a derivation via operational rules with underlying set of direct derivations $\mathcal{D}$.
D is *final* if $\nexists d_{n+1} : d_{n+1} : G_n \overset{cr_{n+1}@cm_{n+1}}{\Longrightarrow} G_{n+1}$ such that d_{n+1} is essential for D.

In Lemma 6.2, we show that for every operation $op \in \{\text{CC, CO, FWD\_OPT, BWD\_OPT}\}$ and every starting triple graph G_0, there exists a final derivation, assuming that the underlying TGG is progressive, such that the process of gathering rule application candidates terminates.

Lemma 6.2 (Existence of a Final Derivation).
Given a progressive TGG for an operation $op \in \{\text{CC, CO, FWD\_OPT, BWD\_OPT}\}$ and a starting triple graph G_0, a final derivation $D : G_0 \overset{*}{\Longrightarrow} G_n$ with operational rules for op exists for every starting triple graph G_0 for op.

Proof. (Sketch) We show the set of essential rule applications according to Definition 6.5 is finite using two arguments: First, the number of possible derivation sequences of some fixed length l is finite for each $l \in \mathbb{N}$ Second, the length of a single derivation sequence consisting only of essential rule applications is also finite.

The first statement can be proven via induction over the length l of the derivation sequence. As all rule applications are essential, they can be sequenced over the $\triangleright^C$ relation by using Condition (2) in Definition 6.5, forming a derivation $d_i \triangleright^C \cdots \triangleright^C d_j$. For the induction step, let $d_i \triangleright^C \cdots \triangleright^C d_j$ be a derivation sequence of length l and let d_k be the next derivation to be added. d_k can only require context elements that are created by any rule application in $d_i \ldots d_j$. According to the induction hypothesis, the number of such possible sequences is finite, so is the number of created context elements, as each rule application can create only a finite number of elements. As only distinct matches are considered (Condition (1) in Definition 6.5), the number of possible sequences $d_i \triangleright^C \cdots \triangleright^C d_j \triangleright^C d_k$ of length $l + 1$ must be finite.

To show that the second statement holds, we derive a contradiction from the assumption that derivation sequence of infinite length consisting solely of essential rule applications can be constructed for a (finite) starting triple graph. The set of essential rule applications can be partitioned into those with and without create dependencies; we will show that both sets are finite. For the former, it can be stated that both the starting triple graph G_0 and the set R of TGG rules are finite. Along with Condition (1) of Definition 6.5 (uniqueness of rule applications), it follows that the set of essential rule applications without create dependencies is finite as well. The TGG is required to be progressive (Definition 6.4), therefore each rule application must mark at least one element. Each element of the starting triple graph G_0 can be marked infinitely often in theory, but Condition (2) in Definition 6.5 prevents an essential rule application d from marking elements that are also marked by a rule application d' if a context dependency $d \triangleright^C_* d'$ exists, stepwise and strictly reducing the set of elements that can be marked by d. As the total number of markable elements in the starting triple graph G_0 is finite, the constructed derivation sequence of essential rule applications must be of finite length.

Combining the arguments for the finiteness of the length of the derivation sequence on the one hand and of the number of sequences of a fixed length on the other hand, the number of essential rule applications must be finite as well, contradicting the assumption and proving the second statement. □

Lemma 6.2 can now be used to show completeness, i.e., that the existence of a consistent solution also implies that there is a final derivation of operational rules from the starting triple graph to this solution.

Theorem 6.2 (Completeness) .
Given a progressive TGG for an operation $op \in \{\text{CC, CO, FWD\_OPT, BWD\_OPT}\}$, and a starting triple graph G_0, a final derivation $D : G_0 \overset{*}{\Longrightarrow} G_n$ for op, a maximal proper subset $\mathcal{D}' \subseteq \mathcal{D}$ and a maximally marked triple graph G' with respect to D exist such that:

$$G' \text{ is a consistent solution} \iff \bigcup_{d \in \mathcal{D}'} mrk(d) = \text{elem}(G_0)$$

Proof. (Sketch) The implication in backward direction of the equivalence follows from Theorem 6.1, such that only the implication in forward direction is to be shown.

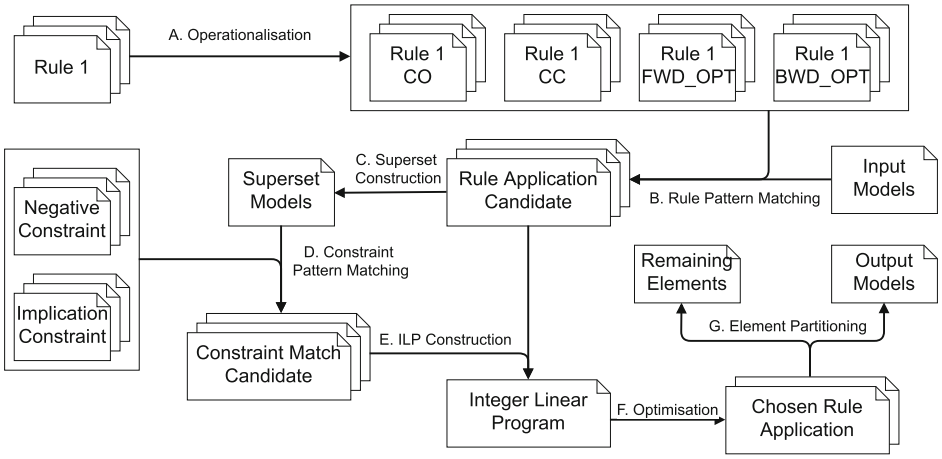


Fig. 8. Workflow for ILP-based consistency management with constraints

To derive a contradiction, we assume that G' is a consistent solution for the given input triple G_0, but G' contains unmarked elements. As G' is consistent, $G' \in \mathcal{L}(TGG)$ and $G' \in \mathcal{L}(TG, \mathcal{GC})$ hold. From $G' \in \mathcal{L}(TGG)$ and the decomposition and composition theorem for TGGs and operational rules [EEE+07, Leb18], if follows that there exists a derivation sequence $D' : G_0 \overset{*}{\Longrightarrow} G'$ via operational rules that entirely marks G_0. From Lemma 6.2, it follows that a final derivation D'' exists. As D' and D'' are not equal, and D'' is the final derivation for which the number of markings is maximised according to the optimisation objective (Definition 5.14), there must be at least one superfluous rule application in D'. This contradicts the assumption $G' \in \mathcal{L}(TGG)$ because superfluous rule applications lead to multiple markings on at least one element, i.e., G' cannot be produced using the respective set of declarative rule applications corresponding to D'. □

7. Implementation

To show that our formal framework is also applicable in practice, it was implemented as part of the Java-based Eclipse plug-in eMoflon::Neo.[2] Unlike most existing MDE tools, eMoflon::Neo is not based on the Eclipse Modelling Framework (EMF) but uses the graph database Neo4J[3] for model querying and persistence. The basic workflow for ILP-based consistency management with constraints is depicted in Fig. 8 as an extension of the process without constraints (cf. [WALS19]).

From the declarative TGG rules, operational rules are derived for the aforementioned consistency management tasks (A). Subsequently, candidates for possible rule applications are determined by graph pattern matching on the input models (B). In eMoflon::Neo, the graph database Neo4j[4] is used to store models and metamodels, such that pattern matching is done by means of generated graph queries. Depending on the operation, the pattern matching step is an iterative process: While one query is sufficient for the CO operation, as operational rules do not create any new elements in this case, multiple iterations are necessary for the other operations to find all rule application candidates. For applying rules, database queries are generated which add new nodes and edges to the graph. The application of all rules from the superset of potential candidates results in "superset models" similar to the example instance of Fig. 5 (C). At this point, the pattern matching step for negative constraints and implication constraints can be performed (D), taking all elements of the input models into account, as well as all elements which were created in the previous step. As a result, a superset of matches for constraint patterns is created, whereby the validity of each match clearly depends on the choice of rule applications.

[2] github.com/eMoflon/emoflon-neo
[3] neo4j.com
[4] https://neo4j.com/

JavaToDoc	# new elements	# rule applications	# created elements
ClazzToDoc	3	1	3
SubClazzToDoc	5	1	5
MethodToEntry	5	4	20
AddParameter	3	4	12
FieldToEntry	5	4	20
AddGlossary	1	0	0
LinkGlossaryEntry	1	4	4
AddGlossaryEntry	2	4	8

Table 1. JavaToDoc: Composition of rule applications for the model generation

FamiliesToPersons	# new elements	# rule applications	# created elements
FamiliesToPersons	3	1	3
MotherToFemale	7	1	7
- *w/o new Family*	5	2	10
FatherToMale	7	1	7
- *w/o new Family*	5	2	10
DaughterToFemale	7	1	7
- *w/o new Family*	5	2	10
SonToMale	7	1	7
- *w/o new Family*	5	2	10

Table 2. FamiliesToPersons: Composition of rule applications for the model generation

Based on the sets of potential rule applications and constraint matches, the ILP can be constructed (E). It consists of an objective function that maximises the number of marked elements and constraints that guarantee language membership and schema compliance. The optimal solution is found by an external solver that is connected to eMoflon::Neo via a generic interface. We used the Gurobi optimizer[5] for our experiments, while an adapter for SAT4J[6] is also implemented. Based on the optimisation solution, the set of actual rule applications can be determined (F). Elements that are marked or created by the chosen rule applications form the output models, whereas - in case of inconsistent inputs - elements that remain unmarked are returned to the user separately (G). Finally, elements that were created by rule applications that are not contained in the optimum solution are deleted, as they contribute neither to the input nor to the result of the operation.

8. Experimental evaluation

In an experimental evaluation we analyse the impact of graph constraints on runtime performance using the example TGG introduced in Sect. 5. In contrast to previous experiments [WA20], all four ILP-based consistency management operations are taken into account. We investigate the scalability of the respective operations for growing model sizes with and without taking graph constraints into account with the following research questions:

(RQ1) By which factor does the number of variables increase for the different operations when introducing graph constraints to the ILP?

(RQ2) How does the runtime performance relate to model size (number of nodes and edges) for consistency management operations with and without graph constraints?

(RQ3) How is the runtime distributed between the different steps, i.e., pattern matching, rule application, ILP construction, and ILP solving?

(RQ4) Do the operations show different scalability characteristics with and without graph constraints?

Setup: As an example TGG, we took the `JavatoDoc` TGG as presented in Fig. 3 along with the constraints as shown in Fig. 2. Furthermore, the bx benchmark example *FamiliesToPersons*[7] was used, extended by additional negative and implication constraints. An overview of the metamodel, the TGG rules and constraints can be found in the appendix (Sect. A).

As inputs for the considered operations, synthetic test models were created with the model generation operation of eMoflon::Neo, which applies a given number of rules randomly, starting with an empty triple. The generated triples had overall model sizes from 720 to 36,000 elements, i.e., nodes and edges. To create models of realistic shape, rule applications were composed with a fixed ratio as shown in Tables 1 and 2. To keep the ratio equal for all model sizes, the number of rule applications was increased with a scaling factor. A factor of 10, for instance, was needed for triples with 720 elements, and a factor of 500 for 36,000 elements.

[5] https://www.gurobi.com/
[6] http://sat4j.org/
[7] http://bx-community.wikidot.com/examples:familytopersons

As a result, the models are guaranteed to be contained in the TGG, but likely violate each of the posed constraints several times. This is desirable, as the impact of taking graph constraints into account on the runtime performance is to be analysed, also for instances with many constraint violations at different points.

Each operation was run on these models (1) without graph constraints, (2) only with negative constraints, and (3) both with negative and implication constraints. Due to the nature of the operations, the entire triple was only given as input to the `CO` operation; the other operations received the respective parts of the triple. The JavaToDoc TGG is not progressive (cf. Definition 6.4) for `FWD_OPT`, such that the rules `AddGlossary`, `AddGlossaryEntry` and `LinkGlossaryEntry` were omitted for this operation. Similarly, the rule `AddParameter` was omitted for `BWD_OPT`. For the FamiliesToPersons TGG, no adaptions were necessary. To reduce the effect of outliers, each configuration was repeated 5 times, and the median was taken as runtime result. Furthermore, the number of binary variables in the ILP was tracked to get an indication for the problem size. The executions took place on a standard notebook with an Intel Core i7 (1.80 GHz), 16GB RAM, and Windows 10 64-bit as operating system. As a prerequisite for eMoflon::Neo, an installation of Eclipse IDE for Java and DSL Developers, version 2021-03 (4.19.0) with Java Development Kit (JDK) version 13 was used. 4GB RAM were allocated to the JVM running the tests, while 8GB were allocated to the graph database Neo4j (version 3.5.8). Gurobi 8.1.1 was used to solve the ILP.

Results:[8] The overall runtime needed for performing the operations on generated models of different sizes is depicted in Figs. 9, 10, 11, and 12, respectively. Note that both axes have a logarithmic scale, making it possible to depict the measurements for small and large model sizes in one diagram. It can be observed that the consumed runtime depends much more on the particular operation than on whether negative constraints are taken into consideration. For all operations, a slightly super-linear increase of runtime can be observed, independent of the consideration of negative constraints. When executing `CO` and `BWD_OPT` for the JavaToDoc example, however, a substantial difference can be observed when taking implication constraints into account as well. An explanation could be that the `CO` operation is quite cheap in general as all relevant rule application candidates can be collected in parallel in a single step, as no new (context) elements are generated during the operation. However, when the operation must handle implication constraints, the constructed ILP gets more complex, which can be observed when considering the increased share of the ILP solving step (Fig. 25). As `Entry` nodes of the documentation model can be transformed to either `Method` or `Field` nodes in the Java model, various possibilities exist for `BWD_OPT` to transform the target model. This could result in an exploding number of premise matches for the `SameNameSameGlossaryEntry` constraint (cf. Fig. 2), leading to a large runtime consumption of the operation.

The number of binary variables moderately correlates with the runtime consumption for both TGGs all operations, reflecting the additional time effort for pattern matching and ILP solving (Figs. 13, 14, 15, 16). Larger differences can only be observed for the FamiliesToPersons example for the operations `CC` and `FWD_OPT` with implication constraints, where the increase in the number of binary variables does not affect the runtime consumption to the same extent. Gurobi's logger messages indicated that it was possible for the solver to substantially reduce the optimisation problem at an early stage, such that pattern matching remained the most costly step for `CC` (cf. Fig. 30) and still played an important role for the overall runtime performance of `FWD_OPT` (cf. Fig. 36). Figures 20–43, which were moved to the appendix for better readability, depict the runtime consumptions of the different phases, namely pattern matching, rule application, ILP construction and ILP solving, for both example TGGs and each of the four consistency management operations. For all operations, ILP solving has a major impact on the runtime performance when handling implication constraints, whereas pattern matching appears to be the most time-consuming step otherwise. Applying the rule candidates to the model triple is expensive for all operations except `CO`, as for this operation, all parts of the triple are given as input and elements need not be created. The time needed for ILP construction is almost negligible, while showing similar scalability characteristics as the solving step.

In total, for negative graph constraints, both the number of variables and the runtime consumption increase by a small factor that remains roughly the same for all model sizes This can be explained with the additional variables as described in Definition 5.7. For implication constraints, both measures substantially increase for most operations and both examples. The root cause is probably the additional complexity which is induced by handling premise and conclusion patterns separately, adding more complexity to the constructed ILP problem.

[8] https://bit.ly/35RAeaz, https://bit.ly/3u5oXgs

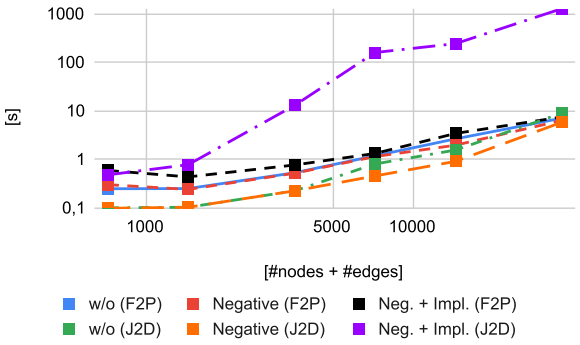


Fig. 9. Runtime: CO

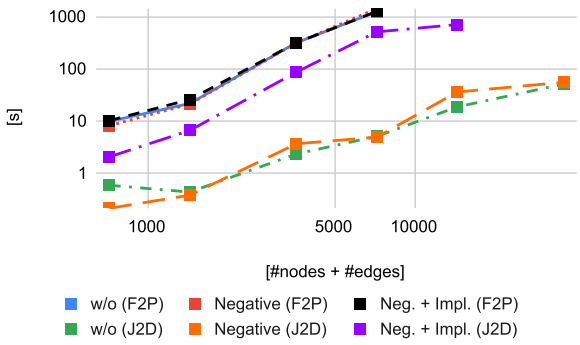


Fig. 10. Runtime: CC

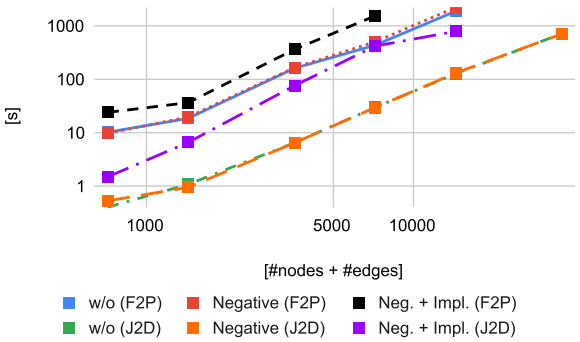


Fig. 11. Runtime: FWD OPT

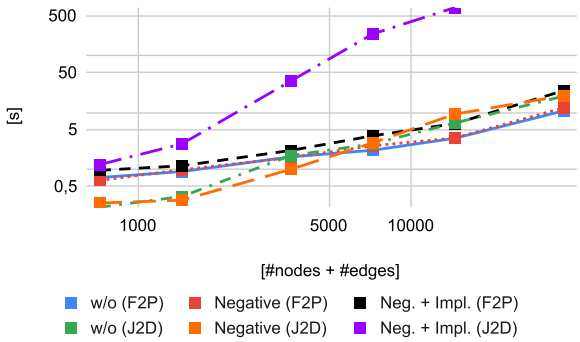


Fig. 12. Runtime: BWD OPT: CC

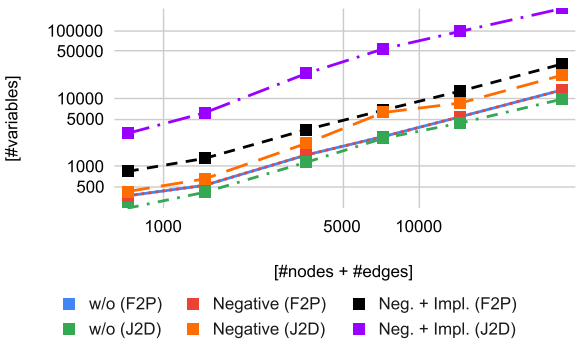


Fig. 13. Number of variables: CO

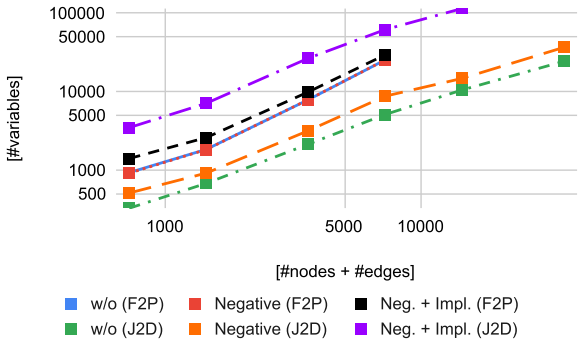


Fig. 14. Number of variables: CC

Summary: Revisiting our research questions, the number of additional binary variables increases moderately for all consistency management operations when adding negative constraints, whereas implication constraints can have a large impact on this measure (RQ1). The time required for each of the steps increases slightly super-linear, whereby there a noticeably larger increase was observable for `CO` and `BWD_OPT` for the JavaToDoc example after adding implication constraints.

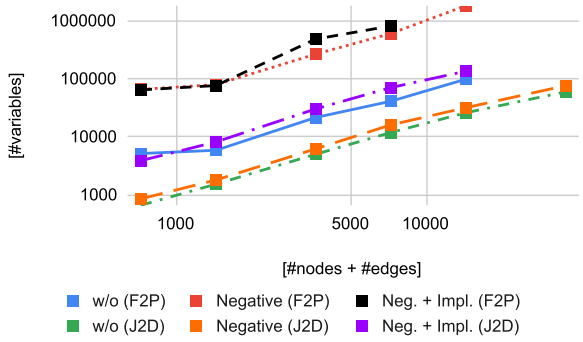


Fig. 15. Number of variables: FWD OPT

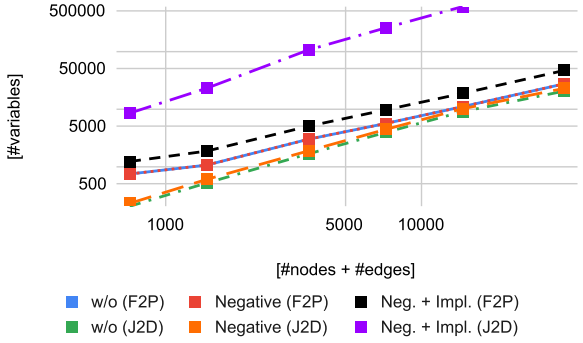


Fig. 16. Number of variables: Number of variables: BWD_OPT

Depending on the TGG and operation, this can become problematic for large model sizes (RQ2). The runtime consumption is dominated by the pattern matching and rule application steps in settings without implication constraints, whereas ILP solving is of major importance otherwise. Efforts for the ILP construction step can almost be neglected, though (RQ3). The runtime performance is much more dependent on the concrete operation than on whether negative constraints are considered, which might also differ depending on the characteristics of the TGG in use. Implication constraints, however, add more complexity to the ILP, whereby the extent again differs between the operations and TGG examples at hand. (RQ4).

Threats to validity: The runtime measurements were conducted only for two TGG, with eight and nine comparably small rules. Also the used graph constraints are composed of rather small patterns. The example models were small- or medium-sized and generated randomly, such that they are not necessarily comparable to realistic use cases. The observed remarkably different runtime measurements for the two TGGs and the four operations indicate that the performance depends on the used TGG, making further tests with other benchmark examples crucial. Our results only hold for Neo4j as graph pattern matcher, and Gurobi as ILP solver. Previous experiments [WA20] have indicated, however, that these two solvers are very well-suited for their respective tasks.

9. Related work

Building upon prior approaches on combining TGGs and ILP [Leb16, LAS17, Leb18, WALS19, WA20], our contribution extends this line of work by formalising graph constraints for (unidirectional) transformation and consistency checking without given correspondences. For the latter, Leblebici et al. initially formalised a hybrid approach of combining consistency checks based on TGGs with ILP as a Search-Based Software En- gineering (SBSE)]technique [Leb16, LAS17], and subsequently proved correctness and completeness [Leb18]. Both the formalisation and the proof of formal properties were generalised for the remaining consistency management operations [WALS19], before graph constraints were introduced to the formal and practical framework, but only supporting consistency checks with given correspondence links [WA20]. Our novel extension guarantees schema compliance for the other ILP-based operations derived from a given TGG and a set of graph constraints.

OCL Integration for Model Transformations: Numerous approaches to model transformation that take additional constraints into account have presented already, partly addressing the issue of constraints for multiple domains. Cuadrado et al. translate constraints of the target model to the source model, such that the result of a forward transformation is guaranteed to satisfy all posed constraints [CGdL+17]. The approach was implemented in the anATLyser tool and applied to real-world model instances. The Object Constraint Language (OCL) is used to define constraints, while the transformation is specified using Atlas Transformation Language (ATL). Compared to our approach, the supported constraints are more expressive, whereas the transformation is unidirectional. The focus is rather set on forward and backward transformations than on supporting a wide range of operations with the same consistency definition. Cabot et al. generate OCL invariants from TGG rules and Query/View/Transformation-Relation (QVT-R) specifications to verify and validate model transformations [CCGdL10]. Both the metamodel and the derived invariants can be used to check whether models are well-formed, which resembles the notion of consistency used in this paper. However, the transformation is decoupled from checking the invariants, whereas our approach integrates both tasks into a uniform algorithm.

Application Conditions for TGGs: To the best of our knowledge, all existing TGG-based approaches ensure schema compliance indirectly by equipping TGG rules with semantically equivalent ACs. Originating from algebraic graph transformation, Ehrig et al. introduced NACs to TGG and proved correctness and completeness for unidirectional model transformation [EHS09]. The formal framework was extended by Golas et al. [GEH11] towards general ACs for TGGs , which reach the expressive power of implication constraints. However, the approach is restricted to the declarative specification of TGGs, enabling the rule-based generation of models that adhere to ACs, whereas an operationalisation is left to future work. In neither of the approaches, the formalisation is at this point backed up by an implementation that could show whether the concepts are applicable in practice. This open challenge was subsequently addressed by Klar et al. [KLKS10], while restricting the class of supported NACs to NACs that are only used to guarantee schema compliance. A translation algorithm with polynomial runtime was presented that proves this class of NACs to be efficiently supported in practice, whereby correctness and completeness of this strategy can still be guaranteed. The semantic equivalence to negative constraints together with a TGG was shown by Anjorin et al. [AST12] by proposing a constructive algorithm for generating such NACs from negative constraints. While the formalisation and implementation was initially restricted to (unidirectional) model transformation, Leblebici et al. [LAF+17] showed evidence that these concepts can be efficiently transferred to (incremental) model synchronisation. Hildebrandt et al. propose a static analysis technique for integrating OCL constraints with TGGs [HLBG12]. The approach is implemented as an extension to the TGG-based model transformation tool MoTE. Transformation and constraint checks are decoupled, and only a subset of the OCL is covered, such that the expressiveness of the supported constraints is equal to those in our approach. Overall, these approaches either support only a subset of ACs or are restricted to a single operation. General ACs, for instance, are only specified for declarative TGG rules, and consistency checks remain totally unaddressed in this regard. Furthermore, all NACs are required to be "domain separable", i.e., restrict the applicability of a rule either for the source or the target model. In contrast, our approach can handle general graph constraints that are also allowed to range over multiple domains including the correspondence model (cf. Fig. 2).

Constraint-based approaches: There are also purely constraint-based approaches that encode both model structure and consistency relation into constraints and can easily handle schema compliance. The JTL framework supports several consistency management operations by deriving constraints from the input models and computing a valid solution via answer set programming [EPT18]. The core.logic[9] of Clojure is used in FunnyQT [Hor17] to perform consistency management operations. The transformation engine Echo is able to derive constraints from metamodels that are enriched with OCL constraints, before handing them over to the Alloy solver [MC13]. The high degree of flexibility of constraint-based approaches, and their potential to support more expressive constraints, such as nested graph constraints, however, comes at the price of scalability, leading to insufficient runtime performance for models of realistic sizes [ABW+19]. Our hybrid approach uses the flexibility of constraint solvers while scaling comparably well [WALS19], as constraints are formed on the level of rule applications, keeping the size of the optimisation problem manageable. In recent work on integrating constraints into algebraic graph transformation [KSTZ20], Kosiol et al. propose to consider consistency as a continuous measure rather than a binary decision. While our approach uses the number of elements in the maximal consistent sub-triple to measure consistency, the authors consider the ratio between all occurrences of a constraint pattern in the model and the number of violations of this constraint. It is shown that graph transformation rules can be classified as consistency sustaining or improving, which means that their application has a non-negative or positive effect on the model consistency. Semeráth and Varró developed a strategy for checking constraints for partial models, i.e., models that involve uncertainty. An early detection of potential or guaranteed violations of well-formedness constraints is implemented via graph pattern matching on the partial model [SV17]. Similarly, different solvers were used to generate consistent models, for which it can be guaranteed that structural and attribute constraints are respected [SBL+20, BSV20, MSBV20]. Both approaches are restricted to intra-model consistency, and the results are not directly transferable to a bidirectional scenario, though.

Search-based Software-Engineering: Linear programming techniques have been combined with pattern matching by Callow and Kalawski [CK13] in a hybrid model transformation approach. However, the Mixed Integer Linear Programming (MILP)-based optimisation aims rather at model compliance than at maximising the number of translated elements, and is tailored to unidirectional transformations instead of taking multiple operations

[9] https://github.com/clojure/core.logic

into account. Xiong et al. solve model synchronisation and consistency management tasks using the Haskell-based language Beanbag [XHZ+09] and compare it to a QVT-R-based synchroniser. While the approach was proven to be correct, the completeness of the transformation is still an open issue. Furthermore, both constraints and correspondences are only implicitly considered. There are also various constraint-based approaches that use evolutionary algorithms or other bio-inspired meta-heuristics and could also handle schema compliance. OCL constraints are combined with graph pattern matching in the tool MOMoT [FTW16] to perform model transformation based on evolutionary algorithms. Design Space Exploration (DSE) as a multi-objective optimisation technique is used by Denil et al. [DJVV14] in combination with the T-core transformation framework [SVL15]. Kessentini et al. combine an example-based model transformation approach with Particle Swarm Optimisation (PSO) as a search technique is used in the tool *MOTOE* [KSB08]. Only about half of the test models were correctly transformed in a case study dealing with the translation of Unified Modeling Language (UML) diagrams into relational databases, though. While heuristic search techniques scale well even for large models sizes, formal properties such as correctness and completeness cannot be proven because the optimality of the returned solution cannot be guaranteed.

10. Conclusion and future work

This paper extends seminal work on ILP-based consistency management with graph constraints. For different consistency management operations, we formalised the handling of domain constraints to guarantee schema compliance, in addition to consistency with respect to the underlying TGG. The approach was implemented in the TGG tool eMoflon for all operations. An experimental evaluation indicated that the introduction of negative constraints does not have a severe impact on the runtime performance, such that small and medium-sized models can be sufficiently handled in real-world applications. Following these promising results, we plan to integrate graph constraints into concurrent synchronisation scenarios. While the formal framework is sufficiently mature and general enough for our purposes, the evaluation clearly suggests that further research is needed on the practical implementation. Further performance tests with benchmark examples and other (industrial) use cases can give insights as to whether the test results are generalisable, and which effect the size of the metamodels, the number and size of TGG rules and other characteristics have on the runtime performance. Finally, we plan to generate graph constraints directly from metamodels, such that the handling of, e.g., multiplicity constraints can be fully automated in the practical framework.

Funding Open Access funding enabled and organized by Projekt DEAL.

A. Appendix

In the following, the metamodel (Fig. 17), TGG rules (Figs. 3, 18) and constraints (Fig. 19) of the benchmark example *FamiliesToPersons* are briefly introduced. In Figs. 20–43, the detailed performance measurements for the different operations, TGGs, and constraint types are presented, with distinct time measurements for the phases of the ILP-based consistency management work-flow.

The FamiliesToPersons example makes use of the rule refinement feature of TGGs [ASLS14]. Rules with abstract types (e.g. `Person` in the target model) can be refined with concrete types to avoid multiple definitions of structurally equivalent rules. In Fig. 18, the two abstract rules at the bottom are refined to concrete rules as stated in Fig. 3. The rule `MotherToFemale` refines `FamilyMemberToPerson` with a `mother` edge pointing from the `Family` to the `FamilyMember` and a `Female` person in the target model. Its version without creating a new Family refines the rule `MemberOfExistingFamilyToPerson`, respectively.

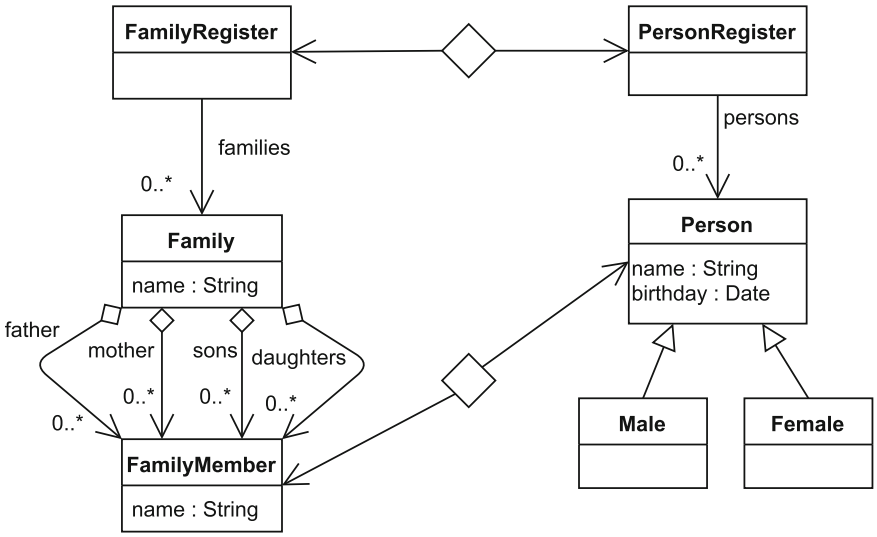


Fig. 17. FamiliesToPersons: Triple metamodel

Table 3. FamiliesToPersons: Rule refinement

FamiliesToPersons	Family → FamilyMember	Person
MotherToFemale	mother	Female
MotherOfExistingFamilyToFemale	mother	Female
FatherToMale	father	Male
FatherOfExistingFamilyToMale	father	Male
DaughterToFemale	daughters	Female
DaughterOfExistingFamilyToFemale	daughters	Female
SonToMale	sons	Male
SonOfExistingFamilyToMale	sons	Male

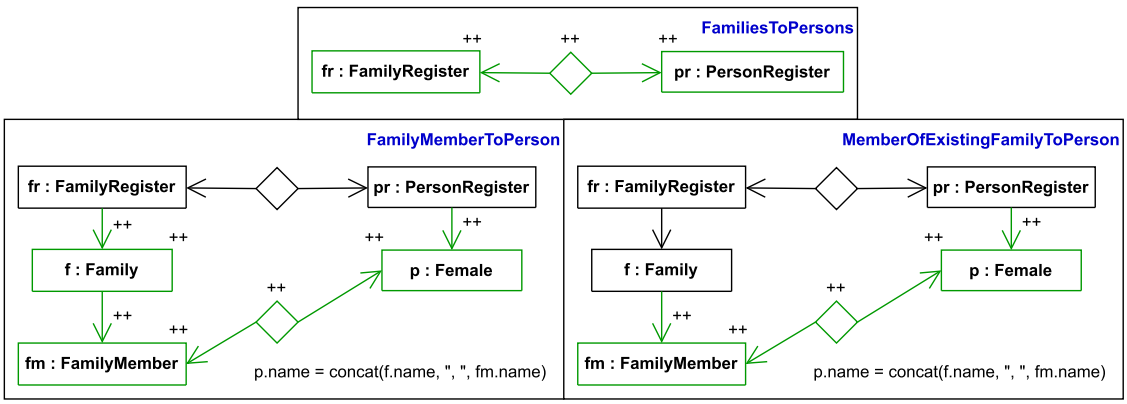


Fig. 18. FamiliesToPersons: TGG rules

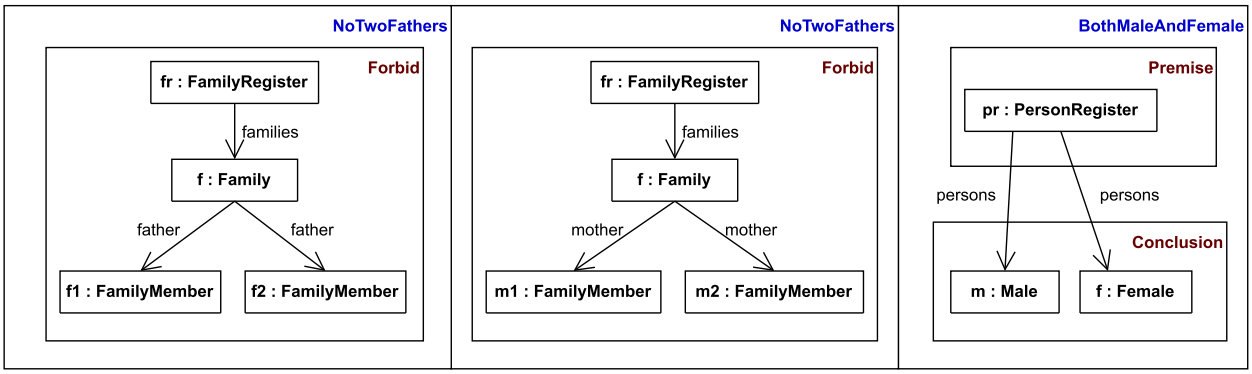


Fig. 19. FamiliesToPersons: Constraints

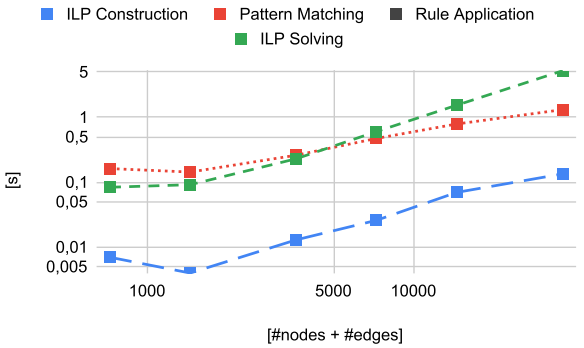


Fig. 20. FamiliesToPersons: CO without constraints

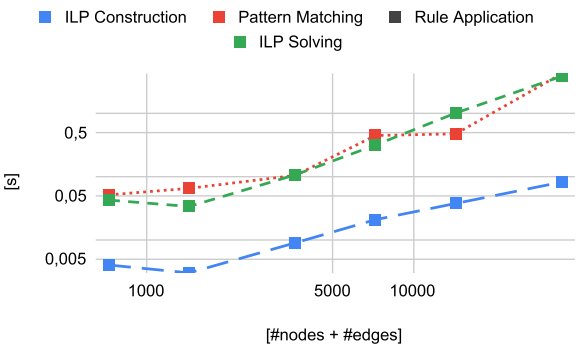


Fig. 21. JavaToDoc: CO without constraints

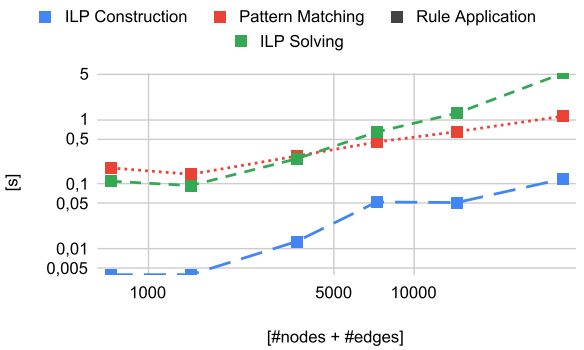


Fig. 22. FamiliesToPersons: CO with negative constraints

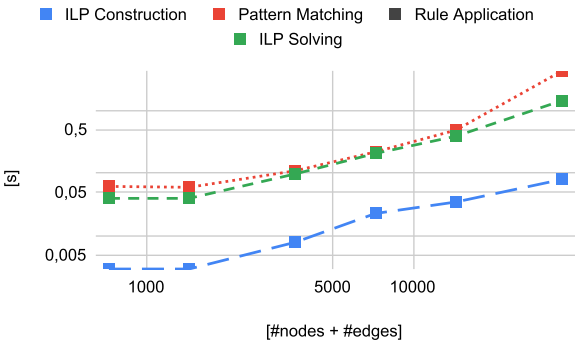


Fig. 23. JavaToDoc: CO with negative constraints

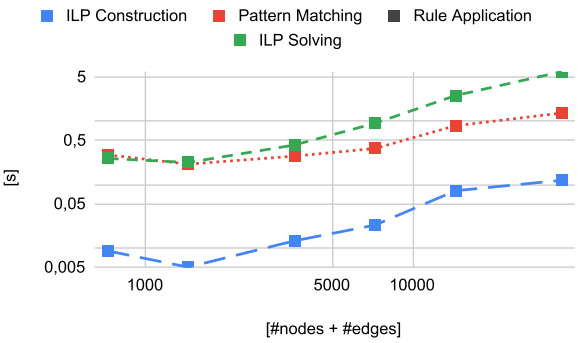


Fig. 24. FamiliesToPersons: CO with implications constraints

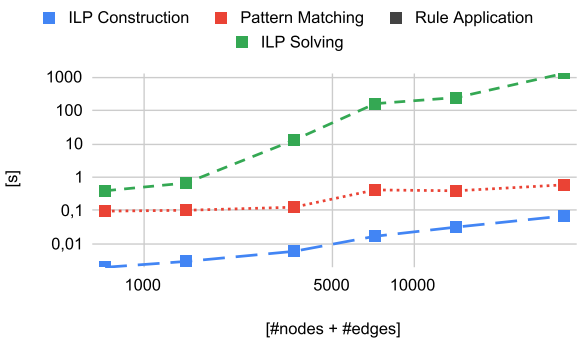


Fig. 25. JavaToDoc: CO with implications constraints

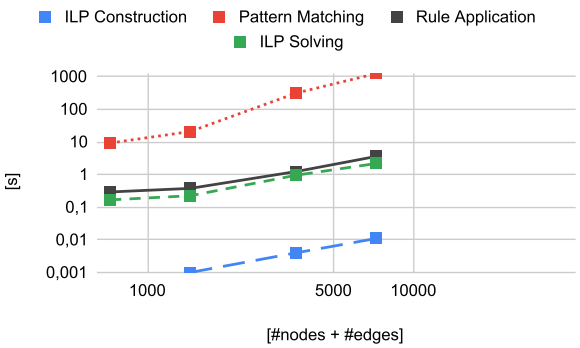


Fig. 26. FamiliesToPersons: CC without constraints

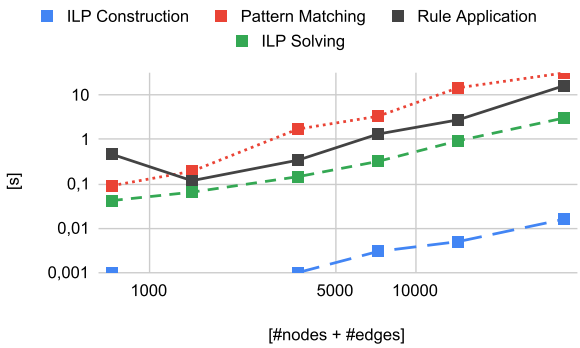


Fig. 27. JavaToDoc: CC without constraints

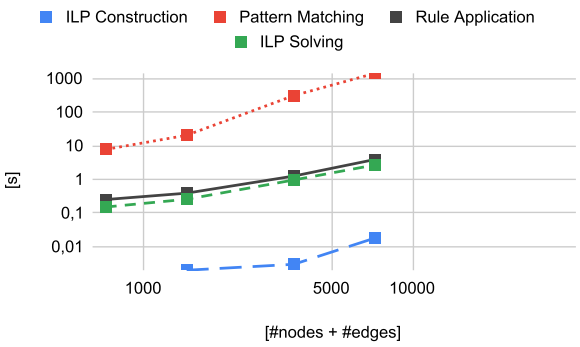


Fig. 28. FamiliesToPersons: CC with negative constraints

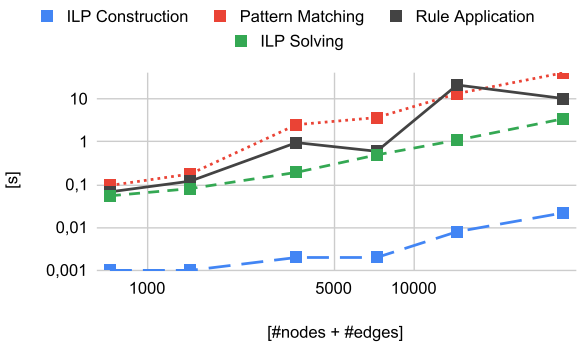


Fig. 29. JavaToDoc: CC with negative constraints

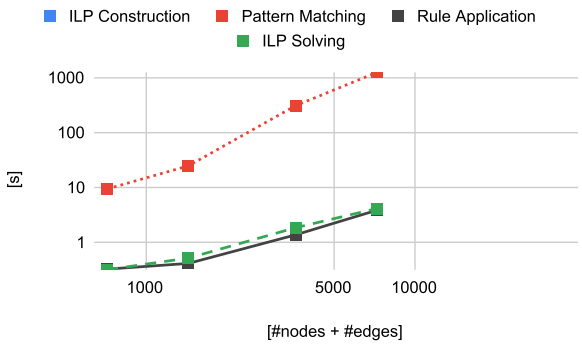


Fig. 30. FamiliesToPersons: CC with implications constraints

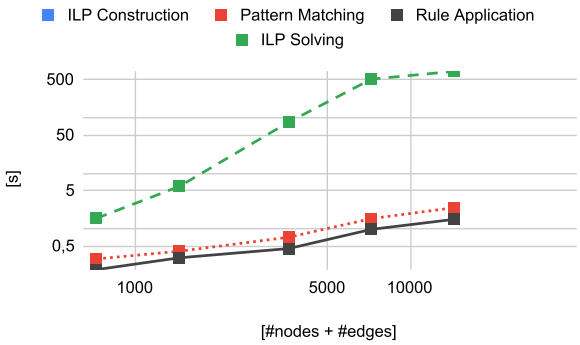


Fig. 31. JavaToDoc: CC with implications constraints

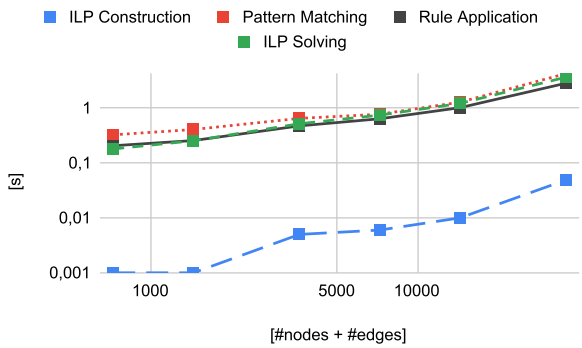


Fig. 32. FamiliesToPersons: FWD_OPT without constraints

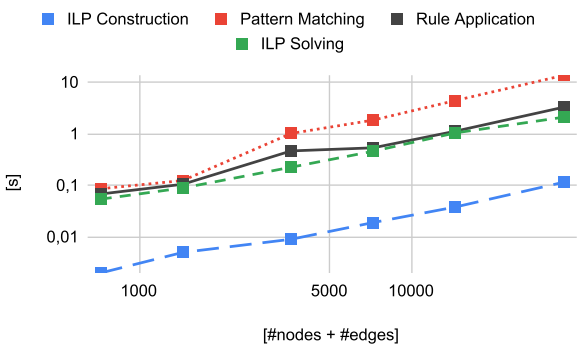


Fig. 33. JavaToDoc: FWD_OPT without constraints

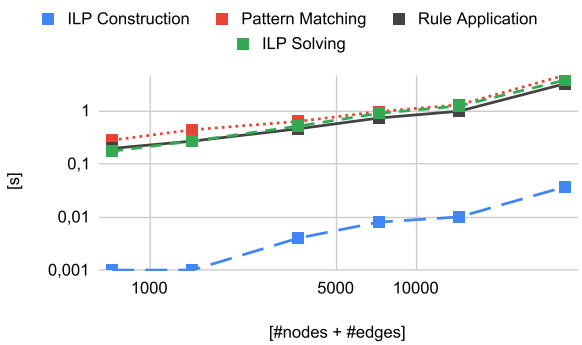


Fig. 34. FamiliesToPersons: FWD_OPT with negative constraints

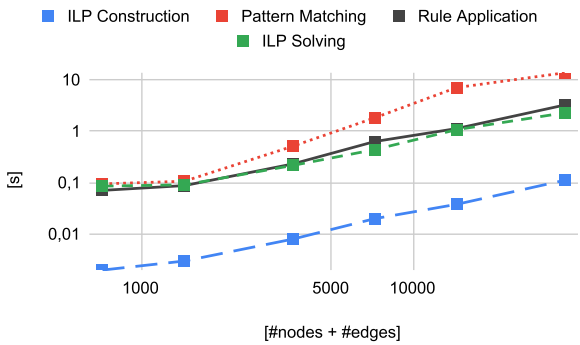


Fig. 35. JavaToDoc: FWD_OPT with negative constraints

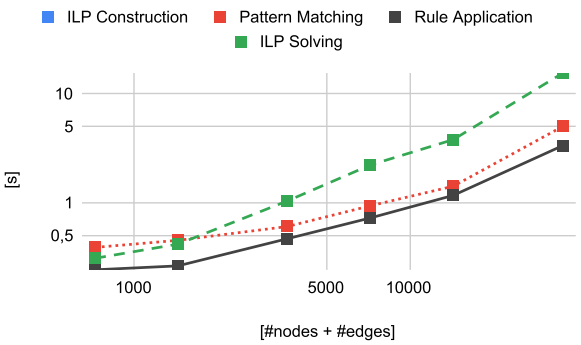


Fig. 36. FamiliesToPersons: FWD_OPT with implications constraints

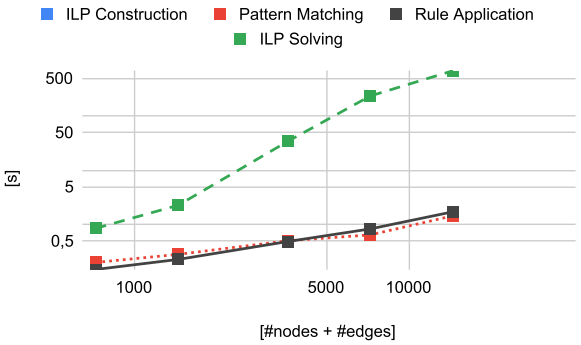


Fig. 37. JavaToDoc: FWD_OPT with implications constraints

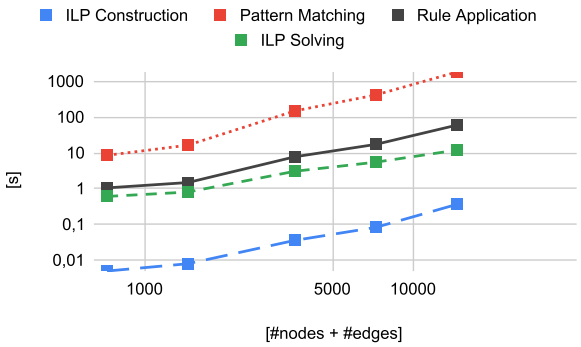


Fig. 38. FamiliesToPersons: BWD_OPT without constraints

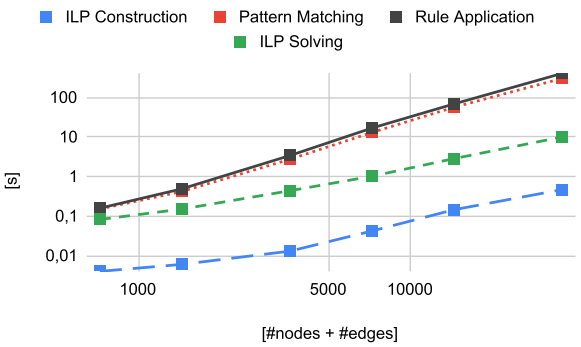


Fig. 39. JavaToDoc: BWD_OPT without constraints

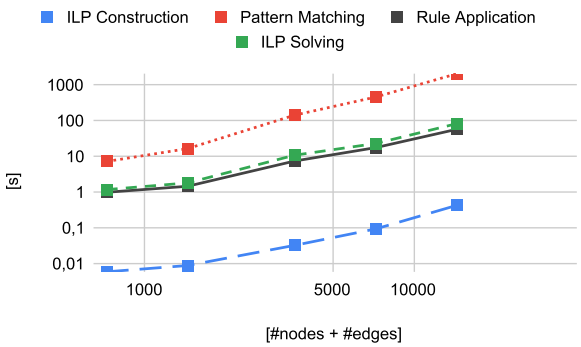


Fig. 40. FamiliesToPersons: BWD_OPT with negative constraints

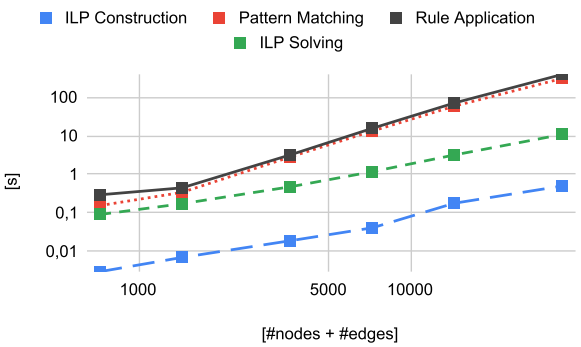


Fig. 41. JavaToDoc: BWD_OPT with negative constraints

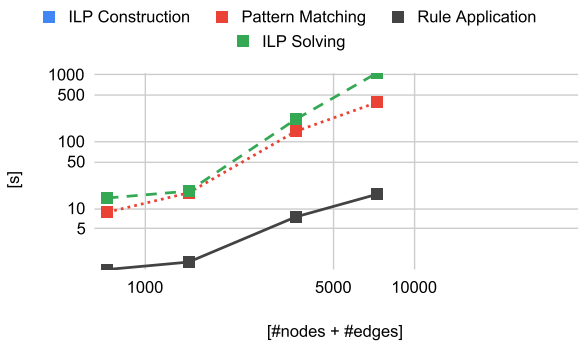


Fig. 42. FamiliesToPersons: BWD_OPT with implications constraints

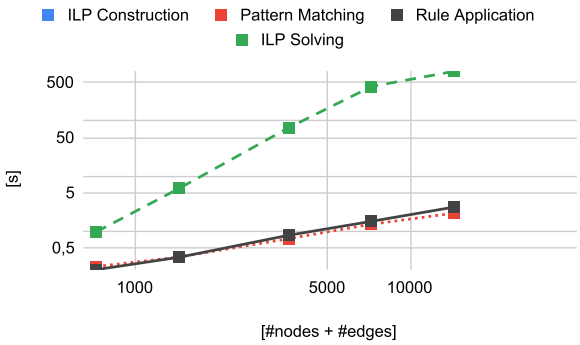


Fig. 43. JavaToDoc: BWD_OPT with implications constraints

References

[AST12] Anjorin A, Schürr A, Taentzer G (2012a) Construction of integrity preserving triple graph grammars. In: Ehrig H, Engels G, Kreowski H-J, Rozenberg G (eds) ICGT 2012. Springer, Berlin

[AVS12] Anjorin A, Varró G, Schürr A (2012b) Complex attribute manipulation in tggs with constraint-based programming techniques. Electron Commun Eur Assoc Softw Sci Technol 49

[ASLS14] Anjorin A, Saller K, Lochau M, Schürr A (2014) Modularizing triple graph grammars using rule refinement. In: Gnesi S, Rensink A (eds) FASE 2014. Springer, Berlin

[ALS15] Anjorin A, Leblebici E, Schürr A (2015) 20 years of triple graph grammars: a roadmap for future research. ECEASST, 73:1–20.

[AYL+18] Anjorin A, Yigitbas E, Leblebici E, Schürr A, Lauder M, Witte M (2018) Description languages for consistency management scenarios based on examples from the industry automation domain. Art Sci Eng Program 2(3):7

[ABW+19] Anjorin A, Buchmann T, Westfechtel B, Diskin Z, Ko H-S, Eramo R, Hinkel G, Samimi-Dehkordi L, Zündorf A (2019) Benchmarking bidirectional transformations: theory, implementation, application, and assessment. In: Software and systems modeling

[BPD+14] Blouin D, Plantec A, Dissaux P, Singhoff F, Diguet J-P (2014) Synchronization of models of rich languages with triple graph grammars: an experience report. In: Di Ruscio D, Varró D (eds) Theory and practice of model transformations—7th international conference, ICMT@STAF 2014, York, UK, July 21–22, 2014. Proceedings, vol 8568. Springer, pp 106–121

[BSV20] Babikian AA, Semeráth O, Varró D (2020) Automated generation of consistent graph models with first-order logic theorem provers. In: Wehrheim H, Cabot J (eds) Fundamental approaches to software engineering—23rd international conference, FASE 2020, held as part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2020, Dublin, Ireland, April 25–30, 2020, Proceedings, volume 12076 of Lecture Notes in Computer Science. Springer, pp 441–461

[CCGdL10] Cabot J, Clarisó R, Guerra E, de Lara J (2010) Verification and validation of declarative model-to-model transformations through invariants. J Syst Softw 83(2):283–302

[CGdL+17] Cuadrado JS, Guerra E, de Lara J, Clarisó R, Cabot J (2017) Translating target to source constraints in model-to-model transformations. In: 20th ACM/IEEE international conference on Model Driven Engineering Languages and Systems, MODELS 2017, Austin, TX, USA, September 17–22, 2017. IEEE Computer Society, pp 12–22

[CK13] Callow G, Kalawsky R (2013) A satisficing bi-directional model transformation engine using mixed integer linear programming. J Obj Technol 12(1):1:1–43

[DJVV14] Denil J, Jukss M, Verbrugge C, Vangheluwe H (2014) Search-based model optimization using model transformations. In: Amyot D, Fonseca i Casas P, Mussbacher G (eds) SAM 2014. Springer, Cham

[EEPT06] Ehrig H, Ehrig K, Prange U, Taentzer G (2006a) Fundamentals of algebraic graph transformation. Springer, Berlin

[EEHP06] Ehrig H, Ehrig K, Habel A, Pennemann K-H (2006b) Theory of constraints and application conditions: from graphs to high-level structures. Fundam Inform 74(1):135–166

[EEE+07] Ehrig H, Ehrig K, Ermel C, Hermann F, Taentzer G (2007) Information preserving bidirectional model transformations. In: Dwyer MB, Lopes A (eds) FASE 2007. Springer

[EHS09] Ehrig H, Hermann F, Sartorius C (2009) Completeness and correctness of model transformations based on triple graph grammars with negative application conditions. ECEASST 18

[EPT18] Eramo R, Pierantonio A, Tucci M (2018) Enhancing the JTL tool for bidirectional transformations. In: Marr S, Sartor JB (eds) Programming 2018, Nice, France, April 9–12, 2018. ACM

[FTW16] Fleck M, Troya J, Wimmer M (2016) Search-based model transformations with MOMoT. In: Van Gorp P, Engels G (eds) ICMT 2016. Springer, Cham

[GEH11] Golas U, Ehrig H, Hermann F(2011) Formal specification of model transformations by triple graph grammars with application conditions. ECEASST 39

[GHN10] Giese H, Hildebrandt S, Neumann S (2010) Model synchronization at work: keeping sysml and AUTOSAR models consistent. In: Engels G, Lewerentz C, Schäfer W, Schürr A, Westfechtel B (eds) Graph transformations and model-driven engineering—essays dedicated to Manfred Nagl on the occasion of his 65th birthday, vol 5765. Springer, pp 555–579

[GHL14] Giese H, Hildebrandt S, Lambers L (2014) Bridging the gap between formal semantics and implementation of triple graph grammars—ensuring conformance of relational model transformation specifications and implementations. Softw Syst Model 13(1):273–299

[HLBG12] Hildebrandt S, Lambers L, Becker B, Giese H (2012) Integration of triple graph grammars and constraints. Electron Commun Eur Assoc Softw Sci Technol 54

[HLG+13] Hildebrandt S, Lambers L, Giese H, Rieke J, Greenyer J, Schäfer W, Lauder M, Anjorin A, Schürr A (2013) A survey of triple graph grammar tools. Electron Commun Eur Assoc Softw Sci Technol 57

[Hor17] Horn T (2017) Solving the TTC families to persons case with FunnyQT. In: García-Domínguez A, Hinkel G, Krikava F (eds) TTC 2017, vol 2026. CEUR-WS.org

[HP09] Habel A, Pennemann K-H (2009) Correctness of high-level transformation systems relative to nested conditions. Math Struct Comput Sci 19(2):245–296

[KLKS10] Klar F, Lauder M, Königs A, Schürr A (2010) Extended triple graph grammars with efficient and compatible graph translators. Springer, Berlin, pp 141–174

[KSB08] Kessentini M, Sahraoui H, Boukadoum M (2008) Model transformation as an optimization problem. In: Czarnecki K, Ober I, Bruel J-M, Uhl A, Völter M (eds) MoDELS 2008. Springer, Berlin

[KSTZ20] Kosiol J, Strüber D, Taentzer G, Zschaler S (2020) Graph consistency as a graduated property—consistency-sustaining and -improving graph transformations. In: Gadducci F, Kehrer T (eds) Graph transformation—13th international conference, ICGT 2020, Bergen, Norway, June 25–26, 2020, Proceedings, vol 12150. Springer, pp 239–256

[Leb16] Leblebici E (2016) Towards a graph grammar-based approach to inter-model consistency checks with traceability support. In: Anjorin A, Gibbons J (eds) Bx 2016. CEUR-WS.org

[Leb18] Leblebici E (2018) Inter-model consistency checking and restoration with triple graph grammars. Ph.D. thesis, Darmstadt University of Technology, Germany

[LAS17] Leblebici E, Anjorin A, Schürr A (2017a) Inter-model consistency checking using triple graph grammars and linear optimization techniques. In: Huisman M, Rubin J (eds) FASE 2017. Springer, Berlin

[LAF+17] Leblebici E, Anjorin A, Fritsche L, Varró G, Schürr A (2017) Leveraging incremental pattern matching techniques for model synchronisation. In: de Lara J, Plump D (eds) ICGT 2017, Marburg, Germany, July 18–19, 2017, Proceedings

[MC13] Macedo N, Cunha A (2013) Implementing QVT-R bidirectional model transformations using alloy. In: Cortellessa V, Varró D (eds) FASE 2013. Springer, Berlin

[MSBV20] Marussy K, Semeráth O, Babikian AA, Varró D (2020) A specification language for consistent model generation based on partial models. J Obj Technol 19(3):3:1–22

[NGdSO19] Nierstrasz O, Gray J, Oliveira BCdS (eds) SLE 2019, Athens, Greece, October 20–22, 2019, Proceedings. ACM

[SBL+20] Semeráth O, Babikian AA, Li A, Marussy K, Varró D (2020) Automated generation of consistent models with structural and attribute constraints. In: Syriani E, Sahraoui HA, de Lara J, Abrahão S (eds) MoDELS '20: ACM/IEEE 23rd international conference on Model Driven Engineering Languages and Systems, Virtual Event, Canada, 18–23 October, 2020. ACM, pp 187–199

[Sch94] Schürr A (1994) Specification of graph translators with triple graph grammars. In: Mayr EW, Schmidt G, Tinhofer G (eds) Graph-theoretic concepts in computer science, 20th international workshop, WG '94, Herrsching, Germany, June 16–18, 1994, Proceedings, vol 903. Springer, pp 151–163

[SV17] Semeráth O, Varró D (2017) Graph constraint evaluation over partial models by constraint rewriting. In: Guerra E, van den Brand M (eds) Theory and practice of model transformation—10th international conference, ICMT@STAF 2017, Marburg, Germany, July 17–18, 2017, Proceedings, vol 10374. Springer, pp 138–154

[SVL15] Syriani E, Vangheluwe H, Lashomb B (2015) T-Core: a framework for custom-built model transformation engines. Softw Syst Model 14(3):1215–1243

[WA20] Weidmann N, Anjorin A (2020) Schema compliant consistency management via triple graph grammars and integer linear programming. In: Wehrheim H, Cabot J (eds) Fundamental approaches to software engineering—23rd international conference, FASE 2020, Dublin, Ireland, April 25–30, 2020, Proceedings, vol 12076. Springer, pp 315–334

[WOR19] Weidmann N, Oppermann R, Robrecht P (2019a) A feature-based classification of triple graph grammar variants. In: Nierstrasz et al. [NGdSO19], pp 1–14

[WALS19] Weidmann N, Anjorin A, Leblebici E, Schürr A (2019b) Consistency management via a combination of triple graph grammars and linear programming. In: Nierstrasz et al. [NGdSO19], pp 29–41

[WFA20] Weidmann N, Fritsche L, Anjorin A (2020) A search-based and fault-tolerant approach to concurrent model synchronisation. In: SLE 2020. Association for Computing Machinery, New York, pp 56–71

[XHZ+09] Xiong Y, Hu Z, Zhao H, Song H, Takeichi M, Mei H (2009) Supporting automatic model inconsistency fixing. In: van Vliet H, Issarny V (eds) Proceedings of the 7th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT international symposium on foundations of software engineering, 2009, Amsterdam, The Netherlands, August 24–28, 2009. ACM, pp 315–324

Received 22 November 2020

Accepted in revised form 12 July 2021 by Jordi Cabot, Heike Wehrheim and Eerke Boiten

Published online 24 August 2021