Software and Systems Modeling (2020) 19:853–888
https://doi.org/10.1007/s10270-020-00791-9

SPECIAL SECTION PAPER

Extending single- to multi-variant model transformations by trace-based propagation of variability annotations

Bernhard Westfechtel[1] · **Sandra Greiner**[1]

Received: 19 July 2019 / Revised: 31 December 2019 / Accepted: 13 March 2020 / Published online: 25 March 2020

Abstract
Model-driven engineering involves the construction of models on different levels of abstraction. Software engineers are supported by model transformations, which automate the transition from high- to low-level models. Product line engineering denotes a systematic process that aims at developing different product variants from a set of reusable assets. When model-driven engineering is combined with product line engineering, engineers have to deal with multi-variant models. In annotative approaches to product line engineering, model elements are decorated with annotations, i.e., Boolean expressions that define the product variants in which model elements are to be included. In model-driven product line engineering, domain engineers require multi-variant transformations, which create multi-variant target models from multi-variant source models. We propose a reuse-based gray-box approach to realizing multi-variant model transformations. We assume that single-variant transformations already exist, which have been developed for model-driven engineering, without considering product lines. Furthermore, we assume that single-variant transformations create traces, which comprise the steps executed in order to derive target models from source models. Single-variant transformations are extended into multi-variant transformations by trace-based propagation: after executing a single-variant transformation, the resulting single-variant target model is enriched with annotations that are calculated with the help of the transformation's trace. This approach may be applied to single-variant transformations written in different languages and requires only access to the trace, not to the respective transformation definition. We also provide a correctness criterion for trace-based propagation, and a proof that this criterion is satisfied under the prerequisites of a formal computational model.

Keywords Model transformation · Software product line · Annotative variability

1 Introduction

This section describes the background of our research (Sect. 1.1), the problem to be addressed (Sect. 1.2), and the proposed solution (Sect. 1.3). Section 1.4 briefly summarizes related work. Section 1.5 explains the contributions of this journal paper compared to the preceding conference paper [47]. Section 1.6 concludes.

1.1 Background

In *model-driven engineering* (*MDE*) [37], high-level *models* [26] are transformed in a series of steps into executable systems. Models are instances of *metamodels*, which are frequently defined with the *Meta Object Facility* (*MOF*) [31]. On the other hand, a wide spectrum of *model transformation languages* has been developed and used [11], even though the Object Management Group issued a standard for model transformation languages [32].

Product line engineering (*PLE*) denotes an organized reuse process for developing a family of *product variants* [33]. In *domain engineering*, a *variability model*—e.g., a *feature model* [22]—is defined, and a *platform* of artifacts is developed. In *application engineering*, a product vari-

Communicated by Richard Paige, Andrzej Wasowski, and Oystein Haugen.

This paper is an extended version of [47].

✉ Bernhard Westfechtel
Bernhard.Westfechtel@uni-bayreuth.de

Sandra Greiner
Sandra1.Greiner@uni-bayreuth.de

[1] Applied Computer Science I, University of Bayreuth, Universitätsstraße 30, 95447 Bayreuth, Germany

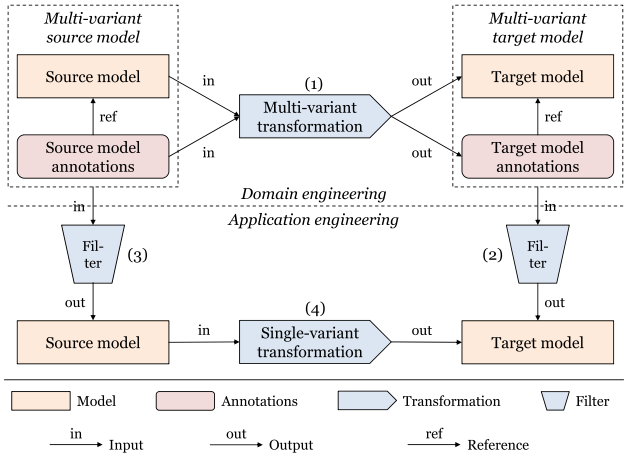

Fig. 1 Commutativity

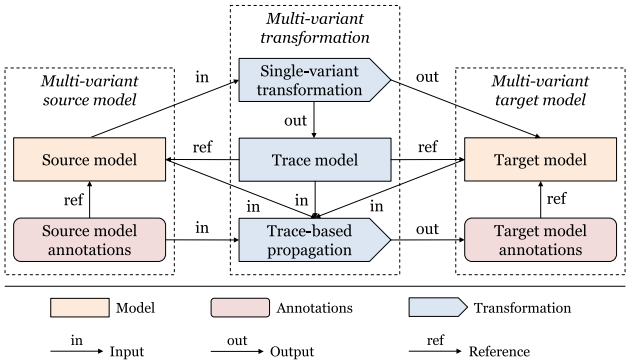

Fig. 2 Trace-based propagation

ant is derived by configuring and adapting the multi-variant platform. In *annotative approaches* [2], artifacts and their elements are decorated with *variability annotations*; a product variant is obtained by filtering all artifacts/elements whose variability annotations evaluate to false (*negative variability*).

Model-driven product line engineering (*MDPLE*) combines MDE with PLE [30]. Thus, models are the artifacts that are subject to variation. In domain engineering, *multi-variant models* are developed that are configured into *single-variant models* by applying *model filters*. Tool support for MDPLE is provided by some commercial tools [5,25] and research prototypes [8,19].

1.2 Problem

When MDE meets PLE, the following problem occurs with respect to model transformations [39]: transformations that have been designed for MDE only ignore variability annotations. Thus, they perform *single-variant (model) transformations* (*SVMT*), i.e., they produce single-variant target models even if they are applied to multi-variant source models. As a consequence, the target model has to be annotated manually, which is both laborious and error-prone.

Instead of variability-ignorant single-variant transformations, *multi-variant (model) transformations* (*MVMT*) [14,15,34] are required that create multi-variant target models from multi-variant source models (upper part of Fig. 1). In this way, the MDPLE user is relieved from annotating targets of model transformations manually.

In addition to automation, multi-variant transformations should meet further requirements. In the first place, the generated annotations should be *correct*, such that there is no need for manual adjustments. A straightforward correctness criterion is *commutativity* [34]: filters and transformations should commute. Thus, a multi-variant transformation (1) followed by a filter on the target model (2) should yield the same result as the same filter applied to the source model (3) followed by a single-variant transformation (4); see Fig. 1.

Furthermore, the effort to realize multi-variant transformations should be minimized. First, single-variant transformations should be *reused*: extending the definition of a single-variant transformation into a multi-variant transformation manually is laborious and error-prone, as well. Second, the realization approach should be *generic*: ideally, it should be language independent, such that it may be applied to transformation definitions written in different languages.

1.3 Solution

We propose *trace-based propagation of variability annotations* [47] as a solution to the problem stated above. Many model transformation tools write traces in addition to target models [6,10,20,28,35,45]. A *trace* records the transformation's execution and is composed of *trace elements* recording which source model elements have been transformed into which target model elements. *Trace-based propagation* is a post-processing approach exploiting the trace to propagate variability annotations from source to target model elements (Fig. 2).

Trace-based propagation reuses the definitions and the results of single-variant transformations as they stand. It is classified as a *gray-box approach* since it requires access to the trace, but not to the transformation definition itself. Furthermore, trace-based propagation is generic because the trace is accessed via a generic interface that is independent of transformation languages and tools.

Filters and transformations commute if single-variant transformations conform to a *computational model* that is characterized as follows: a single-variant transformation is an out-place transformation that operates in batch mode (i.e., the target model is created from scratch from the source model serving as input). A transformation is rule-based; all rules are applied to all matches. Furthermore, rules are monotonic (they only add, but do not modify or delete model elements),

functional (the effect of applying a rule is unique after having fixed the match), and local (only the match is relevant for the rule's application condition and its effects). Finally, traces must be complete, i.e., they must record all relevant elements for each rule application.

1.4 Related work

To the best of our knowledge, trace-based propagation is the only *language-independent approach* to extending single- to multi-variant transformations that has been proposed so far: Trace-based propagation may be applied with any SVMT tool creating a trace—regardless of the respective model transformation language. In contrast, lifting [34] extends an engine for single-variant algebraic graph transformations in such a way that it performs multi-variant transformations. In [40], single-variant transformations written in ATL are extended to multi-variant transformations by a higher-order transformation; the transformation engine itself is reused as it stands. Finally, [15] exploits aspects to extend single-variant transformations defined in Xpand (a template-based model-to-text language). All of these approaches are language dependent.

Furthermore, apart from lifting, trace-based propagation is the only approach for which correctness has been proved (both in this paper and its predecessor [47]).

1.5 Contribution

This journal article is based on a MODELS 2018 conference paper [47]. Compared to its precursor, this extended version includes the following significant changes:

- The first change concerns the presentation. We decided to include a comprehensive introduction to multi-variant model transformations and trace-based propagation. Section 2 summarizes at an informal level the main contributions of our work. In this way, we intend to make the material accessible to a wider readership. Readers are provided with an informal overview of the main concepts and results, without having to delve into the formal part.
- The second change concerns the formal part. In [47], the formalization is *set-based*; models are considered as sets of elements. In the current paper, the formalization is *graph-based*; models are considered as graphs. The formalization follows roughly the same lines as in [47], but it is almost completely new. We consider the graph-based formalization more intuitive and natural than the previous set-based formalization. Furthermore, in contrast to [47] we give a running example in the formal part. Altogether, the new formalization should be more accessible, and it should be easier to compare it against other work on graph transformations.

Fig. 3 Models and model transformations in MDE

1.6 Overview

Section 2 motivates and describes trace-based propagation at an informal level. Section 3 formalizes the computational model and proves commutativity. Section 4 proposes extensions to trace-based propagation that are applied if certain assumptions of the computational model are not satisfied. Section 5 discusses related work. Section 6 concludes.

Readers who are primarily interested in the concepts underlying our approach should focus on Sects. 2 and 4, which are written in an informal style. Readers interested in the formalization should work through Sect. 3, but should at least skim through Sect. 2.

2 Informal description

This section contains an informal description of trace-based propagation of variability annotations. Section 2.1 motivates the need for multi-variant transformations with the help of an example (a graph product line). Section 2.2 deals with multi-variant transformations in general, whereas Sect. 2.3 addresses trace-based propagation in particular. Section 2.4 introduces a simple running example to be used in the subsequent Sect. 2.5, which explains the computational model under which commutativity holds, and in the formalization to be presented in Sect. 3.

2.1 Motivation

In this section, we present an MDE example that motivates the need for multi-variant transformations. The example is given at a conceptual level, without delving into details of the realization.

MDE involves the construction of models on different levels of abstraction such as requirements, design, and implementation (boxes in Fig. 3). The creation of models is supported by model transformations (arrows in Fig. 3). Nonetheless, frequently targets of model transformations still have to be refined manually.

In the following, we focus on the transition from the design model to the implementation model, assuming that the design model is defined by an Ecore class diagram [41] and the implementation is performed in Java. Since the implementation may be derived only partially from the class diagram, it still has to be edited by the user (e.g., by supplying method bodies).

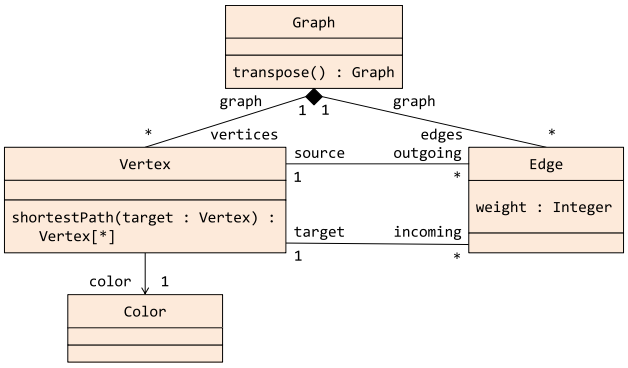

Fig. 4 Design model (Ecore class diagram)

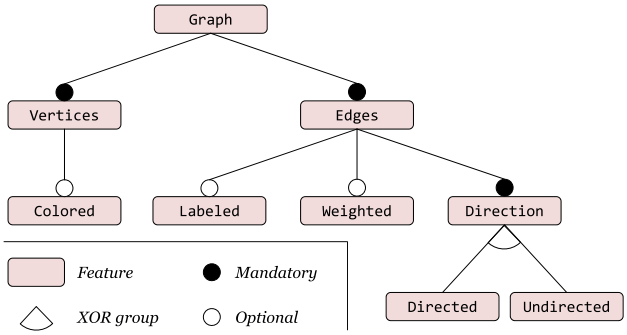

Fig. 6 Feature model

```
public class Vertex {
    protected Color color;
    public Color getColor() {return color;}
    public void setColor(Color color) {this.color = color;}

    protected Graph graph;
    public Graph getGraph() {return graph;}
    public void setGraph(Graph graph) {this.graph = graph;}

    protected List<Edge> outgoing;
    public List<Edge> getOutgoing() {return outgoing;}

    protected List<Edge> incoming;
    public List<Edge> getIncoming() {return incoming;}

    public List<Vertex> shortestPath(Vertex target) {
        // Insert user code
    }
}
```

Fig. 5 Generated source code for class `Vertex`

As a simple example, let us consider a class diagram for *graphs* as shown in Fig. 4. Class `Graph` maintains containment references (decorated with black diamonds) `vertices` and `edges`, both of which are multi-valued (`*`). Both references have an opposite reference `graph` with multiplicity `1`. In contrast to the bidirectional references between `Graph`, `Vertex`, and `Edge`, class `Vertex` holds a unidirectional reference `color` representing the color of the vertex. Class `Edge` maintains an `integer`-valued attribute `weight` for the edge's weight, as well as references `source` and `target` to the edge's end nodes (with opposite references `outgoing` and `incoming`, respectively). Finally, the class diagram contains two operations: `transpose` returns a graph where the sources and targets of edges have been swapped. `shortestPath` returns a path (a sequence of vertices) from the current node to a `target` specified as the operation's parameter.

Let us now assume that source code (in Java) is generated from the class diagram. Figure 5 shows a simplified version of code generated for the class `Vertex`, including fields, `get` and `set` methods with generated bodies,[1] and a method for the shortest path with an empty body, to be supplied by the user. A "real" code generator would generate more sophisticated code, but the details of code generation are not important here.

Since the graph application is well received by the customers, we decide to lift it to a *graph product line* [29] such that we may serve varying requirements with respect to the types of graphs, graph algorithms, graph storage, etc. In *product line engineering (PLE)* [33], domain engineering is distinguished from application engineering. In *domain engineering*, a *variability model* is designed that captures the common and discriminating features of product variants. Furthermore, *multi-variant domain models* have to be developed that reference the variability model to establish mappings between features and their realizations. In *application engineering*, the product line is configured into a specific variant, which may still need to be adapted to specific customer requirements.

Product line engineering approaches may be classified into three categories: *compositional* [4], *transformational* [36], and *annotative* [23]. While compositional and transformational approaches require customized languages for domain artifacts, annotative approaches allow to reuse existing languages for domain models. In the following, we assume an annotative approach to PLE: model elements are decorated with *variability annotations*, which control the sets of product variants in which these elements are included. A product variant is defined by selecting/deselecting features from the variability model. *Single-variant domain models* are obtained by *filtering* those elements whose variability annotations evaluate to false.

Feature models [22] are frequently used to define the variability of product lines. Figure 6 displays the feature model for our graph product line. Features are organized into a tree. `Vertices` and `Edges` are modeled as mandatory features because each graph has vertices and edges. Vertices may be colored, and edges may be labeled or weighted (optional features). Finally, edges are either directed or undirected;

[1] As in EMF code generation [41], we assume that for multi-valued references a `get` method is generated that returns a modifiable list.

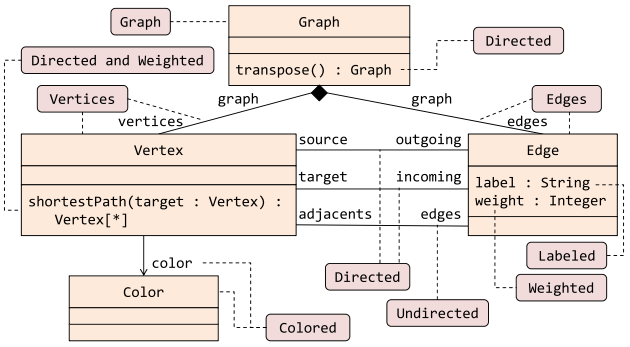

Fig. 7 Multi-variant design model (without multiplicities)

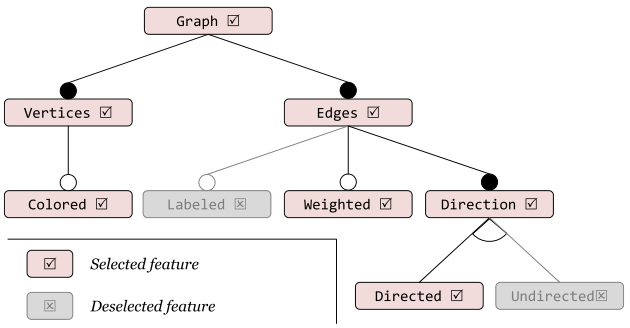

Fig. 8 Feature configuration

```
public class Vertex {
    protected Color color;
    public Color getColor() {return color;}
    public void setColor(Color color) {this.color = color;}

    protected Graph graph;
    public Graph getGraph() {return graph;}
    public void setGraph(Graph graph) {this.graph = graph;}

    protected List<Edge> outgoing;
    public List<Edge> getOutgoing() {return outgoing;}

    protected List<Edge> incoming;
    public List<Edge> getIncoming() {return incoming;}

    protected List<Edge> edges;
    public List<Edge> getEdges() {return edges;}

    public List<Vertex> shortestPath(Vertex target) {
        // Insert user code
    }
}
```

Fig. 9 Result of single-variant model-to-text transformation

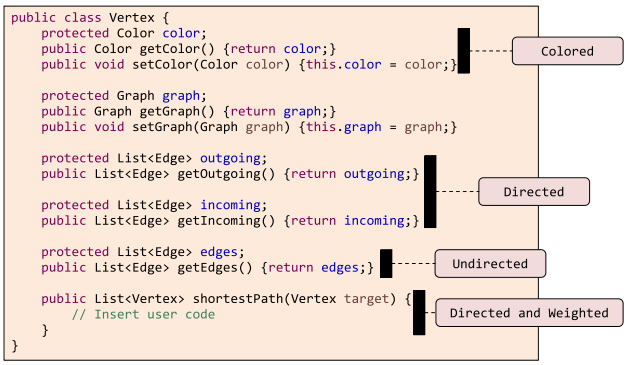

Fig. 10 Manually annotated source code

therefore, the corresponding features are part of an XOR group.

Figure 7 depicts a *multi-variant design model* for our graph product line. Since it constitutes the superimposition of all variants, it is frequently called a *150% model*. Variability annotations, displayed as rounded rectangles, are given as feature expressions. For example, the annotation attached to the operation `shortestPath` states that this operation will be included only in product variants with weighted and directed edges.

A product variant is defined by a *feature configuration*, which defines a selection state for each feature from the feature model (Fig. 8). Each feature is either selected (☑) or deselected (☒). Using the displayed feature configuration, filtering the multi-variant design model of Fig. 7 yields the single-variant design model of Fig. 4.

But what happens when we apply the model-to-code transformation that was used earlier to the multi-variant design model? Figure 9 shows the result. Since the single-variant transformation is *variability ignorant*, it produces a 150% Java class that does not carry any annotations at all. Since each unannotated model element is assumed to be universally included, filtering does not work and merely returns the Java class provided as input. Therefore, the source code returned from the filter will not be consistent with the filtered design model. For example, for the feature configuration from Fig. 8 the source code would include elements for dealing with undirected edges, as well (field `edges` and method `getEdges`, respectively).

Without any automatic tool support, the user has to enrich the source code produced by the single-variant transformation manually with annotations (Fig. 10). For example, elements for undirected edges are annotated with the feature `Undirected`, such that they are removed when this feature is deselected. The user has to ensure *correctness* of annotations: for each feature configuration, the filtered source code must be consistent with the filtered design model. Annotating results of model transformations manually is both laborious and error-prone.

From this sample scenario, we may derive the following problem statement concerning the *integration* of *MDE* with *PLE*: MDE provides *single-variant transformations*, i.e., a single-variant source model is transformed into a single-variant target model. Due to variability ignorance, annotations of the source model are not taken into account. In contrast, PLE requires *multi-variant transformations*, which create multi-variant target models from multi-variant source models. To this end, variability annotations have to be propagated from the source model to the target model. Propagation must be performed correctly, such that each filtered target

model is consistent with the filtered source model (provided that the same filter is applied in both cases).

2.2 Multi-variant model transformations

In the previous section, we motivated the problem to be solved by an example (a graph product line). In the current section, we abstract from this example and consider *multi-variant (model) transformations* at a general level. To this end, we define requirements to multi-variant transformations (Sect. 2.2.1), focusing specifically on correctness (Sect. 2.2.2). After that, we classify and compare different solution approaches (Sect. 2.2.3), including trace-based propagation—the solution proposed in this paper.

2.2.1 Requirements

Tools for multi-variant transformations are utilized in *MDPLE environments*, which provide integrated tools for model-driven product line engineering. From the perspective of a *user* of an MDPLE environment, the following requirements are essential.

Requirement 1 (*Automation*) A multi-variant transformation should propagate annotations from the source to the target model automatically.

As argued in Sect. 2.4, the user should not have to add annotations manually, which is both laborious and error-prone.

Requirement 2 (*Correctness*) A multi-variant transformation should propagate annotations correctly, ensuring that for each filter the filtered target model is consistent with the filtered source model.

Here, consistency is defined by the single-variant transformation: The target model is consistent with the source model if it is equal to the result of a single-variant transformation applied to the source model; see also the next subsection.

From the perspective of a *tool builder* providing an MDPLE environment to the end user, further requirements should be addressed. While the end user requirements stated above seem hardly debatable and should hold in any MDPLE environment, the requirements of the tool builder depend to some extent on the MDPLE environment on which (s)he is working.

Requirement 3 (*Reuse*) Multi-variant transformations should be realized by reusing single-variant transformations.

A multi-variant transformation extends a single-variant transformation inasmuch as it essentially executes the single-variant transformation and propagates variability annotations in addition. Therefore, a tool builder would like to reuse already existing single-variant transformations. Ideally, the single-variant transformation should be reused as it stands—without any need for manual adaptations.

Requirement 4 (*Generic approach*) The problem of extending single- to multi-variant transformations should be addressed in a generic way, minimizing the effort to be invested by the tool builder.

The importance of this requirement depends on the characteristics of the MDPLE environment at hand. At one end of the spectrum, let us consider a *closed environment* with a small set of built-in model transformations, all written in the same language. In such an environment, it may be feasible to extend each of the single-variant transformation definitions manually, by editing the respective transformation definition.

At the opposite end of the spectrum, however, there are *open environments* in which neither model types nor model transformations are fixed. For example, consider MDPLE environments such as Feature Mapper [19] and Famile [8, 9]. Both environments support arbitrary EMF-based domain models and make no assumptions on the tools operating on these models (including model transformation tools and their underlying transformation languages). In environments of this kind, it is crucial to pursue a generic approach to realizing multi-variant transformations.

2.2.2 Correctness

The correctness criterion is defined in terms of *commutativity*: a multi-variant transformation (1) followed by a filter on the target model (2) should yield the same result as a filter on the source model (3) followed by a single-variant transformation (4); see Fig. 1. To the best of our knowledge, this criterion was introduced first in [34].

At first glance, commutativity renders multi-variant transformations obsolete: instead of steps (1) and (2), we may execute steps (3) and (4), returning the same target model. However, in the scenario described in Sect. 2.1, targets of model transformations are domain artifacts that are edited by domain engineers: neither the design model nor the implementation model may be derived completely automatically from the respective master model. For example, the domain engineer has to supplement the source code generated from the design model with hand-written method bodies.

Thus, the domain engineer exploits multi-variant transformations to keep multi-variant models consistent with each other. The transition to application engineering is performed by filtering domain models at all levels of abstraction. Altogether, this development process renders the single-variant transformation (4) in the commuting diagram obsolete—rather than the multi-variant transformation (1). Still, the single-variant transformation may be needed later when changes are performed in application engineering.

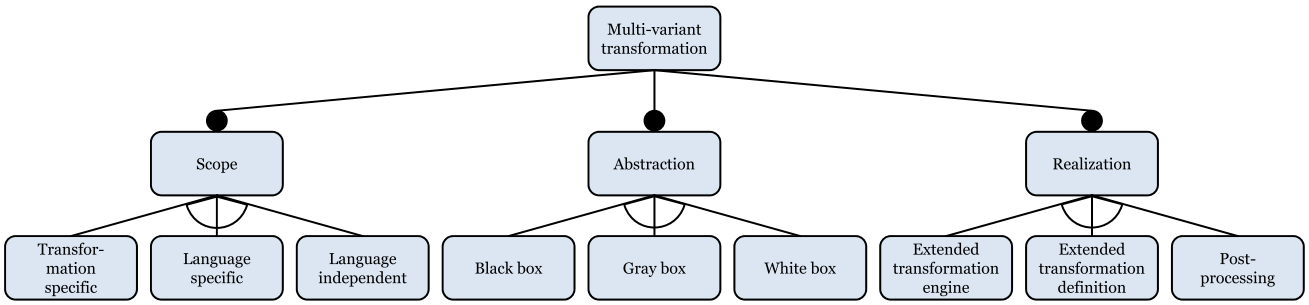

Fig. 11 Taxonomy for multi-variant transformation approaches

Table 1 Classification of MVMT approaches

Approach	Reference	Scope	Abstraction	Realization
Propagation language	[7]	Transformation specific	Black box	Post-processing
Lifting	[13,34]	Language specific	White box	Extended transformation engine
Aspects	[15]	Language specific	Black box	Extended transformation definition
Higher-order transformation	[40]	Language specific	White box	Extended transformation definition
Trace-based propagation	[47]	*Language independent*	*Gray box*	*Post-processing*
Hybrid propagation	[14]	Language specific	White box	Post-processing

2.2.3 Approaches

The problem of extending single- to multi-variant transformations has been addressed in significantly different ways. In the following, we develop a taxonomy and apply it to a set of approaches proposed in the literature. The presentation also includes the approach proposed and elaborated in this paper (trace-based propagation).

Figure 11 shows a feature model for our *taxonomy* of multi-variant transformation approaches.[2] The feature model introduces three dimensions of classification: scope, abstraction, and realization.

The *scope* of a solution delineates the set of transformations to which the solution applies. A solution is *transformation specific* if it is confined to a specific transformation definition. Thus, manually extending the definition of a specific transformation is classified as a transformation specific solution. A solution is *language specific* if it applies to all transformations defined in a specific language. Finally, a solution is *language independent* if the solution may be applied to transformation definitions written in different languages.

The level of *abstraction* defines to what extent the solution makes use of the internals of transformation definitions. A *black box* solution does not consider the transformation definition at all; rather, it assumes only knowledge of its external behavior. A *white box* solution requires full access to the transformation definition. A *gray box* solution is positioned between black box and white box as it requires knowledge of certain aspects of the transformation (to be explained later).

The *realization* dimension refers to the way how a multi-variant transformation is realized. In the case of an *extended transformation engine*, an engine for single-variant transformations is extended such that it performs transformations of multi-variant models. In the case of an *extended transformation definition*, the transformation engine is not modified, rather the definition of a single-variant transformation is extended such that the resulting transformation definition may be executed on multi-variant models. Finally, in the case of *post-processing*, at first the single-variant transformation is executed as it stands, followed by a post-processing step for propagating variability annotations from the source model to the target model.

Table 1 classifies MVMT approaches according to the taxonomy of Fig. 11. Buchmann and Greiner [7] proposes a *propagation language* for specifying the propagation of variability annotations from source to target models. Propagation is performed in a post-processing step. As input to the propagation step, only the source and the target model are provided. This approach is classified as black box because there is no need to access the transformation definition in order to write the propagation rules. Furthermore, it is transformation specific because a propagation definition has to be written for each transformation definition.

Lifting [13,34] extends a transformation engine such that it executes multi-variant transformations. Thus, single-variant transformation definitions need not be changed. The transformation engine provides built-in support for dealing with

[2] A legend for feature models is included in Fig. 6.

variability annotations. This approach is classified as white box because it requires full access to the transformation definition (which has to be executed by the multi-variant engine). Furthermore, it is language specific. In [34] and [13], lifting is applied to in-place algebraic graph transformations [12] and out-place transformations written in DSLTrans [3], respectively.

Instead of modifying the transformation engine, the transformation definition may be extended and then executed by the single-variant transformation engine that is already available. Greiner and Westfechtel [15] introduces a generic *aspect* that may be used to extend any definition of a model-to-text transformation written in the template language XPAND. This approach is language specific, but it is classified as black box because it does not require any knowledge of the actual transformation definition to be extended.

Another approach to extending the transformation definition is described in [40], which proposes a *higher-order transformation*: single-variant transformations defined in ATL are extended into multi-variant transformations by applying a higher-order transformation that yields an extended transformation definition. This approach is language specific and white box because it requires processing of the single-variant transformation definition.

Trace-based propagation [47] exploits the fact that many model transformation tools create traces of the performed transformation steps. In a post-processing step, traces are used to propagate variability annotations (Fig. 2). This approach is classified as gray box: while it does not rely on the actual transformation definition, it does assume that traces are created when the transformation is executed. While each transformation tool uses a specific format (metamodel) for its traces, it turns out that the commonalities of different traces may be abstracted into a language-independent interface. Therefore, trace-based propagation may be performed for all languages and tools that allow to implement this interface.

Figure 12 illustrates trace-based propagation in a simplified example from the scenario introduced in Sect. 2.1. The rectangles placed in the middle are trace elements recording which source elements were transformed into

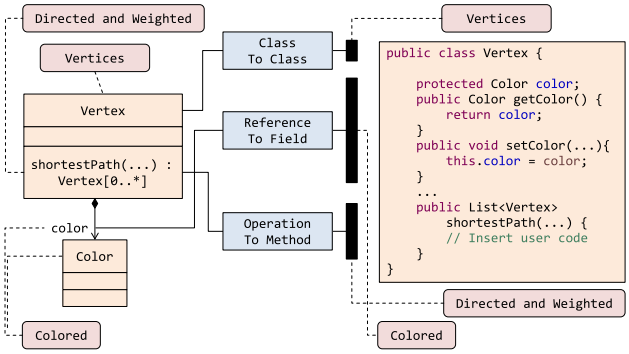

Fig. 12 Example for trace-based propagation

which target elements. By using these mappings, variability annotations are propagated from the source to the target model. For example, the annotation `Directed and Weighted` is copied from the operation computing the shortest path to the corresponding method in the Java class.

Trace-based propagation constitutes a generalization of our previous work referring to transformations written in ATL/EMFTVM. *Hybrid propagation* [14] combines trace-based propagation with an analysis of the byte code model that is generated from a transformation definition and is executed by the virtual machine for ATL/EMFTVM. By means of this analysis, annotations may be propagated even at the level of attributes (which is not possible by analyzing the trace only). In contrast to the work presented in this paper, hybrid propagation is a white box approach which is language specific.

Table 2 summarizes to what extent the approaches described above meet the requirements of Sect. 2.2.1. A tick (✓) and a cross (×) indicate a satisfied and a violated requirement, respectively. Correctness is considered as satisfied only if commutativity has been proved to hold for the respective approach. Only lifting and trace-based propagation satisfy correctness. Only trace-based propagation is classified as generic: it may be applied whenever transformations create traces—regardless of SVMT languages or tools.

Table 2 Satisfaction of requirements

Approach	Reference	Automation	Correctness	Reuse	Generic approach
Propagation language	[7]	✓	×	✓	×
Lifting	[13,34]	✓	✓	✓	×
Aspects	[15]	✓	×	✓	×
Higher-order transformation	[40]	✓	×	✓	×
Trace-based propagation	[47]	✓	✓	✓	✓
Hybrid propagation	[14]	✓	×	✓	×

2.3 Trace-based propagation

In this section, we explain the principles underlying trace-based propagation at an informal level; an illustrating example will be given in the next section. After an overview of trace-based propagation (Sect. 2.3.1), we examine traces in different model transformation tools (Sect. 2.3.2). Based on this analysis, we classify traces (Sect. 2.3.3) and derive an abstract trace metamodel (Sect. 2.3.4). The metamodel serves as an interface that may be realized for different model transformation tools. The generic propagation algorithm is programmed based on this abstract interface (Sect. 2.3.5).

2.3.1 Overview

Many model transformation tools produce traces in addition to target models. A *trace* constitutes a record of the transformation's execution. A trace is composed of *trace elements* recording which source model elements have been transformed into which target model elements. *Trace-based propagation* is a post-processing approach that exploits the trace to propagate variability annotations from source to target model elements (Fig. 2).

2.3.2 Traces

Traces may be structured in different ways. In the following, we examine four model transformation tools that may be considered representative for the types of traces provided by different tools. Subsequently, we abstract from these examples and develop a common trace metamodel as foundation for the propagation algorithm.

medini QVT [20] is based on the OMG language QVT Relations (QVT-R [32]) for uni- and bidirectional model transformations. As far as syntax is concerned, medini QVT conforms to the QVT-R standard but deviates from the semantics definition in the standard. In particular, medini QVT maintains traces of transformations to improve incremental behavior; using traces, changes may be propagated more precisely from source to target models. A trace consists of a sequence of relation instances. Each relation instance records the applied relation as well as the involved source and target elements. The record of a relation instance includes target elements that were created by other relations and serve as context of the current relation application.

ATL [21] is a model transformation language that has been used widely. For ATL, several *virtual machines* are provided that support different types of model transformations. The standard virtual machine included in the ATL distribution may perform both out-place and in-place transformations, but it does not store traces. As far as out-place transformations are concerned, the standard virtual machine performs batch transformations; however, an experimental virtual machine for incremental transformations has been developed, as well [27].

In the context of this paper, we refer to the *ATL/EMFTVM* [45] virtual machine. In contrast to the standard ATL engine, transformations executed in ATL/EMFTVM create a trace that may be analyzed for different purposes. A trace records which rules have been applied to which matches. For each rule application, the trace stores the respective source and target elements. In contrast to medini QVT, only the generated target elements are recorded (not any context elements that may have been required to apply the respective rule). Please note that ATL/EMFTVM supports only a subset of the ATL transformation language; i.e., several restrictions apply compared to the full-fledged ATL language. Furthermore, although ATL/EMFTVM persists traces, incremental transformations are not supported.

BXtend [6] is a framework for bidirectional incremental transformations (written in the object-oriented programming language Xtend) that was inspired partly by triple graph grammars [38]. Similarly to triple graph grammars, a correspondence model is placed between the source model and the target model as an integrating data structure. In its basic version, the correspondence model is composed of 1:1 correspondences (i.e., each correspondence links exactly one source element to exactly one target element). Likewise, context elements are not recorded.

eMoflon [28] provides a language and a set of tools for triple graph grammars. Incremental bidirectional transformations are specified declaratively by triple graph rules. Source and target graph are connected by a *correspondence graph*. Each link node stored in the correspondence graph connects exactly one source to exactly one target node. In addition, eMoflon maintains another data structure that is employed internally for efficiently executing incremental transformations. This data structure is called a *protocol* and maintains a partially ordered set of rule applications. For each rule application, both the match and the created elements are recorded (including nodes and edges of source, target, and correspondence graph). Both the correspondence graph and the protocol may be considered as (different types of) traces.

2.3.3 Classification of traces

Traces may be classified with respect to different criteria. Figure 13 shows a taxonomy for classifying traces as a feature model. Table 3 applies this taxonomy to the trace data structures maintained in the tools that were introduced above. Please note that the table contains two rows for eMoflon because eMoflon maintains two different trace data structures.

With respect to the types of *trace elements*, we distinguish between rule- and link-based traces. A *rule-based trace* records the application of rules. This type of trace is recorded

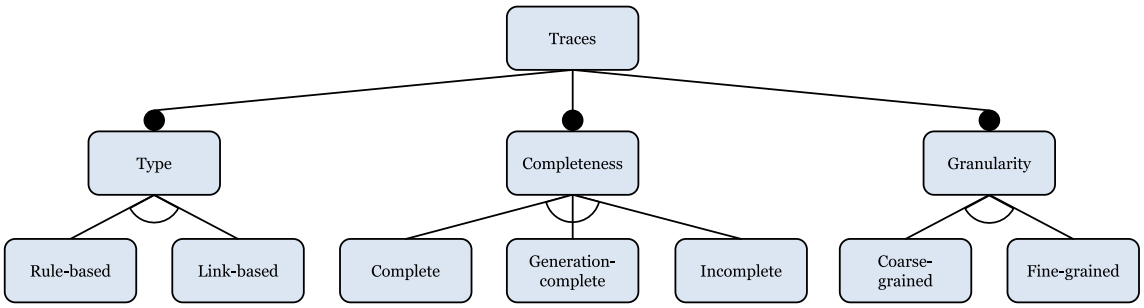

Fig. 13 Classification of traces

Table 3 Traces in different tools

Tool	Trace data structure	Type	Completeness	Granularity
ATL/EMFTVM	Trace model	Rule-based	Generation-complete	Coarse-grained
BXtend	Correspondence model	Link-based	Incomplete	Coarse-grained
eMoflon	Correspondence graph	Link-based	Incomplete	Coarse-grained
eMoflon	Protocol	Rule-based	Complete	Fine-grained
medini QVT	Trace model	Rule-based	Complete	Coarse-grained

automatically by the respective transformation engine essentially in the same way as method invocations are recorded when a program is executed. Each rule instance relates a set of source elements to a set of target elements. The traces maintained by ATL/EMFTVM and medini QVT, as well as eMoflon's protocols, belong to this category.

In contrast, a *link-based trace* is composed of trace links, each of which relates a set of target elements to a set of source elements from which the target elements were created. In contrast to a rule-based trace, a link-based trace has to be managed explicitly in the transformation definition. The traces maintained by BXtend and the correspondence graphs of eMoflon belong to this category. For example, eMoflon's TGG rules have to specify the construction of the correspondence graph explicitly.

Completeness constitutes another dimension of classification (see also Fig. 18 in Sect. 2.4). In a *complete trace*, each trace element records all required and created source, target, and context elements. Both medini QVT and the protocols recorded by eMoflon provide complete traces. A *generation-complete trace* covers only the created target elements and the source elements from which they were generated, but no context elements that may have been required for executing the corresponding transformation step. The traces of ATL/EMFTVM belong to this category since they include all generated target elements. Finally, an *incomplete trace* includes only primary target elements, from which further target elements may be deduced (e.g., by including only the root of some subtree). Both the correspondence graphs of eMoflon and the links maintained by BXtend may be incomplete. Please note that completeness and type are orthogonal criteria: completeness refers to the amount of information that is recorded in the trace; type refers to the way a trace is created (implicitly for rule-based and explicitly for link-based traces).

Finally, we may classify traces with respect to the *granularity* of source and target elements. A *coarse-grained trace* is maintained at the level of objects; a *fine-grained trace* takes object properties (attributes and links) also into account. eMoflon's correspondence graphs, the correspondence model of BXtend, and the traces maintained by ATL/EMFTVM and medini QVT are classified as coarse-grained because they record only object-to-object relationships. The protocols recorded by eMoflon are more fine-grained because they include both objects and links (but no attribute values).

Trace-based propagation may be applied to all kinds of traces. However, it works best for rule-based, complete, and fine-grained traces inasmuch as the proof of commutativity requires traces of this kind (see Sect. 2.5). Only complete and fine-grained traces provide sufficient information for accurate propagation of variability annotations. Furthermore, the computational model on which the proof of commutativity is based assumes rule-based traces.

2.3.4 Trace metamodel

From the traces realized in different tools (Sect. 2.3.2), we abstract a *generic trace metamodel* that may be realized in any of these tools (Fig. 14). We assume that both the source and the target model are composed of model elements, leaving open whether these are objects (coarse-grained traces) or object properties (fine-grained traces). A *trace model* stores

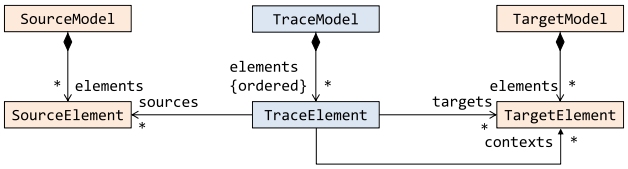

Fig. 14 Trace metamodel

an ordered set of *trace elements*, where each trace element corresponds either to a rule application (rule-based trace) or to a link (link-based trace). Each trace element holds three references: `sources` returns the set of *source elements*, while `targets` and `contexts` are used to determine the sets of created *target elements* and required *context elements*, respectively. A given element of the target model may play the role of a target element only once (it is created only once), while it may serve multiple times as a context element. The reference `elements` is assumed to be ordered in such a way that an element of the target model occurs as a context element only after it has occurred as a target element. The order is relevant only in the case of complete traces (in generation-complete or incomplete traces, each trace element has an empty set of context elements).

The metamodel introduced above is *minimal* in the sense that it includes only the information that is required for trace-based propagation. For this reason, the notion of context element is defined only for the target model. Here, it is crucial to distinguish between context and target elements: the propagation algorithm takes the annotations of context elements (and source elements) as inputs and assigns annotations to target elements. For the sake of orthogonality, the same distinction could be made on the source model: a context element in the source model would be an element that has been transformed by a preceding rule; the elements to be transformed in the current rule would be denoted as source elements. However, trace-based propagation does not require this distinction on the source model: all elements of the source model are treated in the same way—regardless of whether they have been transformed by a previous rule or are transformed in the current rule. Therefore, the trace metamodel handles all source elements in a uniform way.

2.3.5 Propagation algorithm

Based on the trace metamodel introduced above, a *propagation algorithm* transfers variability annotations from source model elements to target model elements. A sketch of this algorithm follows below, whereas a formal description is given in Sect. 3.8.

Trace elements are processed sequentially according to the order of the reference `elements`. For some trace element, let SE, TE, and CE denote the sets of its source, target, and context elements, respectively. Furthermore, let us assume an attribute *ann* for storing annotations of source and target model elements (without prescribing any physical realization; an annotation may be stored either in the respective model or in a separate mapping model). Finally, let $te \in TE$ denote any created target element. To each target element, the conjunction of the annotations of all source and context elements is assigned:

$$te.ann := \left(\bigwedge_{se \in SE} se.ann\right) \wedge \left(\bigwedge_{ce \in CE} ce.ann\right). \quad (1)$$

Equation 1 assumes that the presence of any target element depends on the presence of all source and context elements. Thus, the target element is visible (passes a filter on the target model) if and only if all source and context elements are visible, as well.

Below, we discuss in detail under which conditions this algorithm ensures commutativity (Sects. 2.5, 3). For now, we would like to mention two conditions that refer to the properties of traces introduced in Sect. 2.3.3 above. First, the trace should be *sufficiently fine-grained*: if traces are recorded at the level of objects but annotations are assigned at the level of object properties (e.g., attribute values), the propagation algorithm cannot take variability annotations of object properties into account. Second, the trace should be *complete*: Variability annotations can be determined correctly only when the annotations of all required context elements are taken into account. If the trace is not sufficiently fine-grained or it is not complete, the propagation algorithm may assign incorrect variability annotations.

2.4 Example

In this section, we introduce a running example to be used throughout the rest of this paper. The example [46] is taken from *project management* [24] and involves a transformation from Gantt diagrams to CPM networks. In contrast to the model-to-text scenario introduced in Sect. 2.1, this transformation involves only a small number of rules. For this reason, the running example is suited better for demonstrating trace-based propagation, in particular, with respect to the formalization of our approach (Sect. 3).

Gantt diagrams and CPM networks are widely spread notations for project planning. A *Gantt diagram* (Fig. 15, left) consists of activities (time-scaled bars) and dependencies (arrows). An *activity* is composed of a name and a duration (represented here as a non-negative integer value). A *dependency* connects a *predecessor* to a *successor* activity and is decorated by an offset (an integer that we assume to be non-negative). Dependencies are classified into four types: start–start, start–end, end–start, and end–end. For example, an end–start dependency (as shown in Fig. 15) implies that

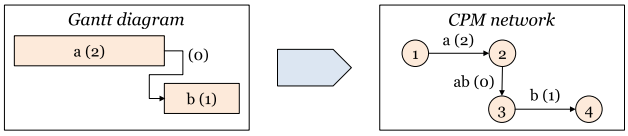

Fig. 15 Gantt diagram to CPM network (concrete syntax)

the successor may start only after its predecessor has finished and its offset has elapsed.

A *CPM network* (Fig. 15, right), which is based on the Critical Path Method, is composed of events and activities. An *activity* has a name and a duration (a non-negative integer) and connects a source event to a target event. An *event* does not consume time and may occur only after its incoming activities have finished. An activity is performed only after its source event has occurred.

In a *Gantt to CPM transformation*, each Gantt activity is mapped to a CPM activity with the same name and duration, and two events acting as source and target of the activity. Each Gantt dependency is mapped to a CPM activity that connects already existing events. The name of the CPM activity is composed of the names of the predecessor and successor activities in the Gantt diagram. The duration is copied from the offset of the dependency. The source and target events are determined from the dependency type. For example, in the case of an end-start dependency the target event of the predecessor activity is connected to the source event of the successor activity.

So far, we have presented models in *concrete syntax* as they are represented at the user interface. Now, we switch to *abstract syntax*, i.e., the models' internal representation in terms of objects and links. In the following, we consider models as *graphs*, which are composed of *nodes* decorated with *attributes*, and binary, directed *edges* connecting these nodes. Both nodes and edges are typed.

Figure 16 shows the Gantt diagram (left) and the CPM network (right) from Fig. 15 in abstract syntax. Nodes are represented as rectangles with an inscription of the form $n : T$, where T denotes a node type and n is an arbitrary unique node identifier. Each graph has a *root node* that is connected to the *element nodes* contained in this graph by *els* edges. Activity nodes are decorated with attributes n and d for the name and the duration, respectively. Dependency nodes are typed by the respective dependency type (e.g., ES for end-start) and are attributed with an offset o. An activity is connected to outgoing dependencies by $a2d$ edges; edges from dependencies to activities are labeled with $d2a$. Similarly, events and activities are connected by $e2a$ and $a2e$ edges in the CPM network.

In its middle, Fig. 16 shows a *trace* from a single-variant transformation. Numbers inside the trace nodes indicate their position in the trace sequence. Node types may be interpreted either as link types (link-based traces) or rules (rule-based

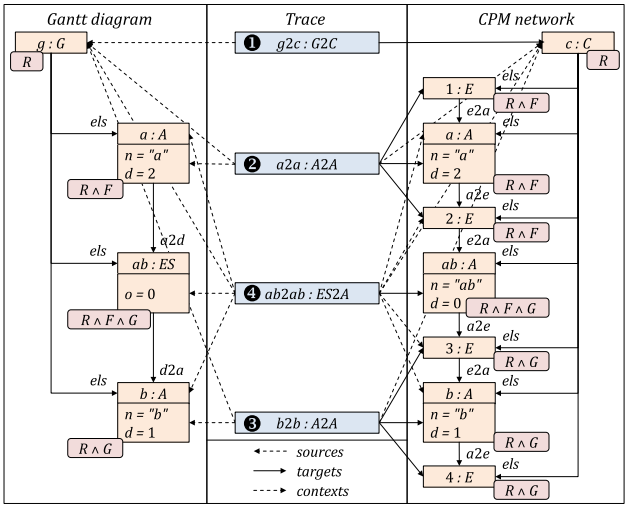

Fig. 16 Gantt diagram to CPM network (abstract syntax)

traces). According to the trace metamodel of Fig. 14, we distinguish between edges to source nodes, target nodes, and context nodes. To keep the figure legible, edge types are not represented by labels; rather, edges are distinguished by orientation and line style.

Here, we assume a *complete trace* according to the classification of Sect. 2.3.3. Trace node (1) maps the root node of the Gantt diagram to the root node of the CPM network. Trace nodes (2) and (3) map activity nodes onto each other. In addition to the activity nodes, the root nodes are included as context nodes (please recall that our trace metamodel distinguishes between target nodes and context nodes in the target model; no distinction between different roles of nodes is necessary in the source model). Finally, trace node (4) maps a dependency node to an activity node (which is possible only after the activity nodes have been mapped). To ensure completeness, the trace node is linked to the root node and the nodes for predecessor and successor activities in the source graph, and to the root node, the corresponding activity nodes, and the connected event nodes in the target graph.

Figure 16 also shows variability annotations. To simplify our running example, we assume *coarse-grained annotations* referring to nodes, and *coarse-grained traces* with node-to-node mappings. The underlying feature model (not shown in the figure) consists of a root feature R and two optional subfeatures F and G. Annotations are defined such that existential dependencies are taken into account. Thus, each element node carries an annotation that implies the annotation of the root node; likewise, the annotation of the dependency node implies the annotations of the nodes for the predecessor and successor activity, respectively. By using trace-based propagation, annotations are added to the target graph as shown on the right-hand side of the figure.

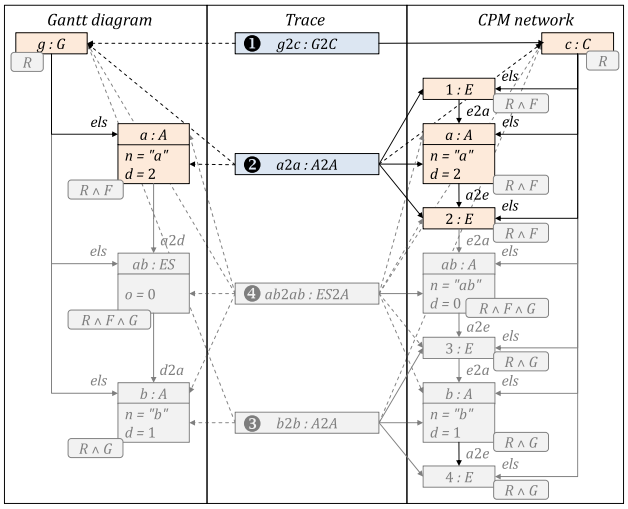

Fig. 17 Filtering under the feature configuration $R \wedge F \wedge \neg G$

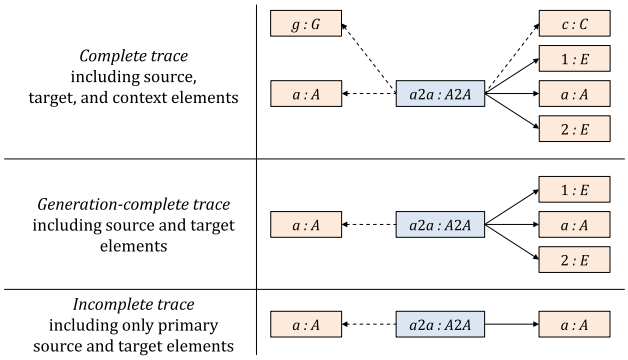

Fig. 18 Completeness of traces

Figure 17 shows the result of filtering the source and the target graph by the same feature configuration $R \wedge F \wedge \neg G$. All parts that are displayed in gray have been removed from the graphs. Note that we assume that filtering retains referential integrity at the level of graphs: if a node is removed, all of its attributes and adjacent edges are removed, as well. In the source graph, only root node g, activity node a, and the edge from g to a pass the filter. Accordingly, only root node c, activity node a, event nodes 1 and 2, and the respective connecting edges remain in the target graph.

The reader may easily check that trace-based propagation achieves commutativity in this example: transforming the filtered source graph yields the same target graph as obtained by filtering the multi-variant target graph. Effectively, the trace has been filtered, as well: only those trace nodes are retained for which all adjacent source, context, and target nodes are still present.

To conclude this section, let us discuss the impact of the completeness of traces on the quality of the resulting propagation (Fig. 18). As already mentioned above, a complete trace is required to ensure commutativity. A generation-complete trace records only the created target elements and the source elements from which they were generated. For mapping activities, this means that the root nodes are not recorded as context of the mapping. In our example, generation-complete traces result in the same variability annotations as for complete traces. As long as the variability annotations of context elements are implied by annotations of source and target elements, commutativity is still achieved. However, with an incomplete trace it is no longer possible to achieve commutativity. For mapping activities, an incomplete trace would record only the generated activity node in the CPM network. The source and target events may be inferred from the activity node by navigating adjacent edges. However, trace-based propagation propagates an annotation only to the activity node and ignores event nodes. As a consequence, event nodes are universally visible and are not removed under any feature configuration.

2.5 Commutativity

After having explained the principles of trace-based propagation at an informal level, we return to the issue of correctness raised in Sect. 2.2.2. Thus, we investigate under which conditions trace-based propagation guarantees commutativity of filters and transformations, as visualized in Fig. 1. This question will be answered with a formal proof in Sect. 3. In the current section, we give a brief summary of the formalization at an informal level, with the intent to convey the main ideas underlying the formalization.

Section 2.5.1 introduces a computational model under which trace-based propagation guarantees commutativity. Section 2.5.2 gives an illustrative example. A sketch of proof follows in Sect. 2.5.3. A discussion of the computational model (Sect. 2.5.4) concludes this section.

2.5.1 Computational model

The *computational model* refers to single-variant transformations and will be formalized in Sect. 3. It is defined such that the following proposition holds (to be proved later):

> If a single-variant transformation conforms to the computational model, trace-based propagation guarantees commutativity of filters and transformations.

In the following, we summarize the *key properties* of the computational model.

Properties of transformations

Property 1 (Out-place transformation) *A transformation is performed* out-place*, i.e., it creates a target model from a source model.*

Thus, the source model and the target model are different, and the source model is not modified.

Property 2 (Batch transformation) *A transformation is performed in* batch mode*, i.e., the target model is created from scratch.*

The computational model deals with *batch transformations*, as they are assumed in the commutativity criterion. *Incremental transformations*, which propagate changes from a source model to an already existing target model, are beyond the scope of the computational model (but are discussed briefly in Sect. 4.4).

Property 3 (Rule-based transformation) *A transformation is composed of* rules*. It is executed by applying all rules to all matches exactly once.*

Since a trace essentially records the application of rules, we may assume that it is generated by a rule-based transformation. This assumption holds even if the trace is link-based provided that there is a 1:1 correspondence between rules and links. Furthermore, the computational model assumes that all rules are applied to all matches. In particular, explicit *control structures* are not covered by the computational model (rules are organized into a global, implicit loop).

Properties of rules

Property 4 (Functional behavior) *If a rule is applied to a given match, the result is determined uniquely.*

In rule-based languages, rules usually exhibit functional behavior. In the context of the computational model, functional behavior carries over from individual rules to transformations as a whole. If a single-variant transformation exhibits non-functional behavior, filters and transformations may not commute because a transformation may render different results in different executions.

Property 5 (Monotonicity) *Each rule is* monotonic*: application of a rule only adds elements to the target model.*

Deletions or modifications are not taken into account by the computational model. The trace metamodel (Fig. 14) assumes that elements in the target model are either reused as context elements or created anew.

Property 6 (Locality) *Each rule is* local*: if a rule is applied to a match, its effect depends only on the match.*

Thus, the context of the match in the source model and the target model having been created so far is immaterial to the execution of the rule. Only then may we assume that a rule is applicable to a given match in a filtered model if and only if it is applicable to the same match in an unfiltered model. In the case of a positive application condition referring to the context of the match, a rule that is applicable in an unfiltered model may not be applicable in the filtered model. Conversely, a negative application condition may prevent a rule from being applied in an unfiltered model even though it is applicable in a filtered model. Therefore, we have to exclude application conditions going beyond the match itself.

Properties of traces

Property 7 (Complete traces) *The rule-based transformation must leave a complete trace behind, recording for each step the applied rule as well as all source, context, and target elements.*

The issue of completeness has already been discussed at the end of Sect. 2.4. If target elements are missing, they will not receive variability annotations, rendering them universally visible. If source or context elements are missing, propagated annotations may be too wide, violating commutativity, as well.

Property 8 (Fine-grained traces) *The trace of a rule-based transformation must be recorded on a fine-grained level.*

In our running example (Sect. 2.4), we assume annotations and traces on object level to simplify matters. As to be shown in the formalization (Sect. 3), coarse-grained traces are not sufficient to guarantee commutativity. In general, the annotations of all involved model elements—including objects, attribute values, and links—are required to calculate the annotations of target elements correctly.

2.5.2 Example

The computational model to be formalized in Sect. 3 satisfies all properties listed above. To illustrate the computational model, we resume the running example introduced in Sect. 2.4. Thus, we present rules for transforming Gantt diagrams to CPM networks. These rules are applied in a loop until all matches are exhausted. In this way, an out-place batch transformation is performed.

To simplify matters, we still assume object-level traces, but we will consider fine-grained traces in the formalization of the rules in Sect. 3. It is assumed that traces are recorded automatically, including source, context, and target elements (complete traces). Thus, creation of the trace is not specified in the rules. However, traces may be accessed in the rules. This approach, which is applied, e.g., in medini QVT and ATL/EMFTVM, simplifies the definition of rules.

For representing rules, we use an informal *graphical notation* that will be mapped to the formalism for the computational model in Sect. 3. A rule is decomposed into three regions: *source* (including source elements), *trace* (including previously created trace elements), and *target* (including

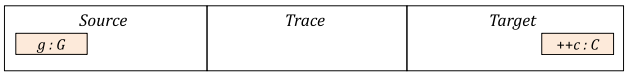

Fig. 19 Diagram root to network root

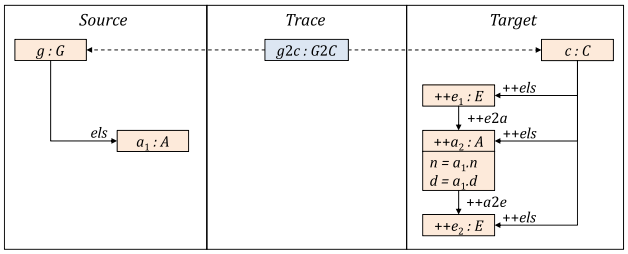

Fig. 20 Activity to activity

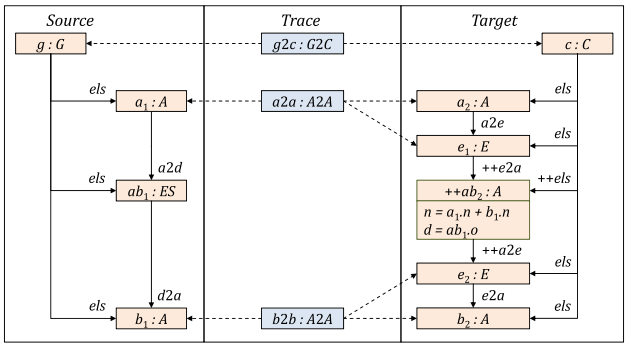

Fig. 21 End-start dependency to activity

both context and created elements). Created nodes and edges are decorated with two plus signs; all other nodes and edges are matched. Equations in matched nodes define conditions on their attribute values, while equations in created nodes define assignments to attribute values.

The rule for mapping the single *diagram root* (Fig. 19) matches the diagram root in the source graph and creates a network root in the target graph. The trace region is empty because no trace element exists before applying this initial rule.

The rule for mapping an *activity* (Fig. 20) requires that the diagram root containing that activity has been mapped to the network root. In the target graph, an activity is created along with its source and target events and connecting edges. The name and the duration are copied from the corresponding activity in the diagram.

The rules for mapping dependencies vary with respect to the dependency type. Here, we present only one example [mapping of an *end-start dependency* (Fig. 21)]. The type of the dependency node must be end-start. A corresponding activity is created in the network, along with edges from the network root as well source and target events being selected based on the dependency type. For an end-start dependency, the target event of the predecessor activity acts as source event for the new activity, and the source event of the successor

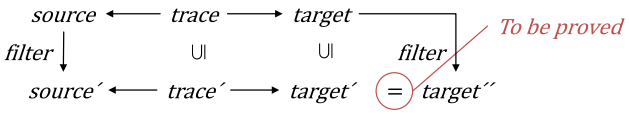

Fig. 22 Commutativity proof

activity plays the role of a target event for the new activity. The trace is used to ensure that the diagram root has been mapped to the network root, and predecessor and successor of the dependency have been mapped to corresponding activities in the CPM network. Finally, the duration of the new activity is copied from the offset of the dependency, and the name is obtained by concatenating (+) the names of predecessor and successor activities.

When the rules presented above are applied to the Gantt diagram shown on the left-hand side of Fig. 16, the CPM network on the right-hand side is created along with the trace displayed in the middle.

2.5.3 Sketch of proof

For the sketch of the proof, which will be elaborated in Sect. 3, we use Fig. 22 as illustration. On the *transform-filter path*, an (annotated) target model $target$ is created from an annotated source model $source$, leaving the trace $trace$; subsequently, $target$ is filtered, resulting in the filtered target model $target''$. On the *filter-transform path*, filtering yields the filtered source model $source'$, which is subsequently transformed into $target'$, producing the trace $trace'$. For proving commutativity, we have to show $target' = target''$.

For the computational model described in Sect. 2.5.1, transformations are monotonic in the following sense: if $source'$ is included in $source$, $target'$ is included in $target$, and $trace'$ is included in trace. Let us consider a step in the transformation of $source'$. If some rule has a match in $source'$ and the part of $target'$ that has been constructed so far, the same rule also has a match in $source$ and a part of $target$ that includes $target'$. Since a rule is local, its applicability depends only on the match, not on its context. Thus, the corresponding step is executable in the transformation of $source$. Since all rules are applied to all matches, every step of the transformation of $source'$ may be mapped to a corresponding transformation step on $source$. Since rules are functional, both transformation steps have the same effect.

The computational model also implies that transformations exhibit functional behavior. Since rules are monotonic and local, two rules that are applicable simultaneously may be executed in any order, giving the same result (confluence). Since all rules are applied to all matches, transformations differ only in the order in which rules are applied to their matches, but not in the final result. Furthermore, it can be shown that if a transformation terminates on some model

Table 4 Satisfaction of properties of the computational model in different tools

Tool	Properties of transformations			Properties of rules			Properties of traces	
	Out-place	Batch	Rule-based	Functional	Monotonic	Local	Complete	Fine-grained
ATL/EMFTVM	×	✓	×	✓	✓	×	×	×
BXtend	✓	×	×	✓	×	×	×	×
eMoflon	✓	×	×	✓	✓	×	✓	✓
medini QVT	×	×	×	✓	✓	×	✓	×

source, it will terminate on any model *source*′ that is included in *source*.

Let us assume that the transformation of *source* terminates. Then, it delivers a unique result *target*, and any transformation of a filtered source *source*′ terminates and returns a unique result *target*′, as well. The equality of *target*′ and the filtered target *target*″ may be proved as follows: let us consider the first step in the transformation of *source*. The respective rule may be applied to the same match in *source*′ if and only if all elements of the match pass the filter. In this case, trace-based propagation ensures that elements that are generated by this step in *target* pass the filter, as well (Eq. 1). Thus, the same elements are generated on both paths. This argument extends by induction to all transformation steps.

2.5.4 Discussion

If a single-variant transformation conforms to the computational model introduced in Sect. 2.5.1, trace-based propagation guarantees commutativity. Thus, the computational model is useful to understand under which conditions filters and transformations commute.

In the following, we examine to what extent model transformation languages and tools satisfy the properties of the computational model introduced in Sect. 2.5.1. Table 4 summarizes the results for the languages/tools introduced earlier in the section on traces (Sect. 2.3.2).

In the table, a tick ✓ indicates that the respective property is guaranteed to hold for every transformation defined and executed in the respective tool. In contrast, a cross × means that the respective property may be violated.

The discussion below is organized by the properties introduced in Sect. 2.5.1, followed by a brief summary.

Out-place transformation (Property 1) All tools support out-place transformations. ATL/EMFTVM and medini QVT go beyond the computational model because they support in-place transformations in addition. Thus, Property 1 may be violated in ATL/EMFTVM and medini QVT, while it is guaranteed to hold in BXtend and eMoflon.

Batch transformation (Property 2) For out-place transformations, only ATL/EMFTVM is constrained to batch transformations and thus conforms to Property 2 of the computational model. All other tools support incremental transformations (considering batch transformations as a special case), which again breaks the computational model.

Rule-based transformation (Property 3) In all languages, transformations are defined in terms of rules. Nevertheless, Property 3 is violated in all cases: all rules need to be applied to all matches in a global loop. ATL/EMFTVM, BXtend, and medini QVT break this computational model because they support explicit control structures and allow rules to be called from other rules. Due to the absence of control structures, eMoflon comes closest to the computational model. However, even eMoflon does not conform to Property 3: eMoflon does not apply all rules to all matches but attempts to cover the source model by rule applications. Thus, if two rules compete for the same match, only one of them is applied.

Functional behavior (Property 4) In all languages, rules are functional, i.e., they have a unique result once a match has been fixed. However, functional behavior of individual rules may not carry over to transformations as a whole. For example, in eMoflon transformations may exhibit non-functional behavior (due to different orderings of rule applications and different resolutions of conflicts). As we have already mentioned in Sect. 2.5.3, our computational model implies functional transformations.

Monotonicity (Property 5) eMoflon's TGG rules are monotonic since they specify synchronous extensions of source, correspondence, and target graphs. Similarly, in medini QVT rules declaratively specify relations among patterns, without offering any language constructs for state changes (updates or deletions). ATL/EMFTVM imposes several restrictions concerning the use of the ATL language. Since updates and deletions may not be specified in rules executable by the ATL/EMFTVM virtual machine, rules are classified as monotonic, as well. BXtend's rules violate monotonicity; arbitrary updates and deletions may be programmed in the object-oriented language Xtend.

Locality (Property 6) Locality is a crucial property of the computational model: only then may a rule be applicable to a given match in an unfiltered model if and only if it is applicable to the same match in a filtered model. In all of the considered approaches, locality is violated: BXtend permits arbitrary BXtend code, checking non-local conditions. Likewise, the use of OCL in ATL/EMFTVM and medini QVT allows to break locality. Finally, in eMoflon negative application conditions allow to specify non-local rules (whose applicability depends on the context of the match).

Complete traces (Property 7) By default, traces in BXtend are incomplete because each correspondence connects exactly one primary source element to exactly one primary target element (see also Fig. 18). In ATL/EMFTVM, traces are generation-complete and do not include context elements. medini QVT's traces are complete since they include all objects involved in the respective relations. Finally, the rule-based protocols recorded by eMoflon are complete, as well (while the link-based correspondence graphs are incomplete).

Fine-grained traces (Property 8) In ATL/EMFTVM, BXtend, and medini QVT, traces are recorded at the level of objects; thus, they are classified as coarse-grained. Only the protocols created by eMoflon include fine-grained elements: all matched or created nodes and edges are recorded. Thus, annotations attached to edges may be taken into account by trace-based propagation. However, node attributes are not recorded. Phrased in the terminology of object-oriented modeling, protocols partially include structural features of objects (links between objects); thus, they are classified as fine-grained.

Summary The preceding discussion shows that none of the considered tools satisfies all properties of the computational model. This means that it is possible to define transformations that break the computational model. However, the large number of crosses in Table 2 might be misleading: when the languages and tools are used in restricted ways, the resulting computations may satisfy the required properties.

For example, let us consider eMoflon: this tool may perform batch transformations by executing an incremental transformation against an initially empty target model. Furthermore, TGG rules that do not include negative application conditions are local. Finally, as long as rules are non-conflicting, all rules are applied to all matches (rule-based transformation).

If a single-variant transformation conforms to the computational model, trace-based propagation guarantees commutativity. Thus, the computational model is useful to understand under which conditions filters and transformations commute. Even if commutativity is violated, trace-based propagation may still be applied in a heterogeneous MDPLE environment. Then, a post-processing step may improve the result of trace-based propagation (see also Sect. 4, which presents extensions to trace-based propagation).

3 Formal description

This section formalizes trace-based propagation. It includes a formal proof of commutativity that is sketched in Sect. 2.5. While the formalization constitutes a key contribution of this paper, it may be skipped by readers who are interested primarily in concepts and applications of trace-based propagation.

After some basic definitions (Sect. 3.1), the first part of this section introduces a *computational base model* (Sects. 3.2–3.5). Sections 3.2 and 3.3 define graphs and graph morphisms (type- and structure-preserving mappings), respectively. Section 3.4 introduces rules that perform graph transformations *in-place*. These rules are functional, monotonic, and local. Furthermore, this section also defines derivations, in which all rules are applied to all matches. Finally, several properties of derivations that are needed for the proof of commutativity are stated and proved in Sect. 3.5.

The computational base model relies on in-place transformations. In Sect. 3.6, we simulate *out-place batch transformations generating traces* with the help of in-place transformations on graphs that are composed of source, trace, and target graphs. Altogether, the resulting *source-to-target transformations* satisfy all properties stated in Sect. 2.5.

The computational model introduced so far is confined to single-variant transformations. Section 3.7 is concerned with variability. In particular, *multi-variant graphs* are introduced, in which each graph element is decorated with an annotation defining in which variants the respective element is included.

Section 3.8 presents an *algorithm* for *trace-based propagation* that is executed after a single-variant source-to-target transformation. The algorithm receives a graph consisting of a source graph, a trace graph, and a target graph as well as annotations of the source graph as inputs and returns annotations of the target graph as output. Altogether, a multi-variant transformation is executed by chaining a single-variant transformation and trace-based propagation.

Finally, Sect. 3.9 presents a formal proof of commutativity: for the graph-based computational model, it is proved that a multi-variant transformation, followed by a filter on the target graph, returns the same result as a filter on the source graph, followed by a single-variant transformation.

3.1 Basic definitions

This section collects definitions that are based on elementary set theory [18] and are used throughout the rest of Sect. 3.

Let S and T be *sets*. We use conventional set-theoretic notation to denote the *union* ($S \cup T$), *disjoint union* ($S \dot\cup T$), *intersection* ($S \cap T$), and *difference* ($S \setminus T$) of S and T. The *inclusion* of S in T is denoted by $S \subseteq T$, and $\emptyset$ denotes the *empty set*. Finally, for a finite set S its *cardinality* (the number of its elements) is denoted by $|S|$.

For sets S and T, the *Cartesian product* $S \times T$ is defined as the set of all pairs (s, t) where $s \in S$ and $t \in T$. A *relation* over S and T is a set $R \subseteq S \times T$. The *domain* of R is defined as $dom(R) = \{s \in S \mid \exists t \in T : (s, t) \in R\}$. The *range* of R is defined as $ran(R) = \{t \in T \mid \exists s \in S : (s, t) \in R\}$.

$R \subseteq S \times T$ is a (total) *function* if $dom(R) = S$ and $(s, t) \in R$ and $(s, t') \in R$ implies $t = t'$ for all $s \in S$ and $t, t' \in T$. Using conventional functional notation, $f : S \to T$ denotes a function f from S to T. For a finite set S, we write f as a set of mappings $\{s_1 \mapsto t_1, \ldots, s_n \mapsto t_n\}$, where $f(s_i) = t_i (1 \leq i \leq n)$.

f is *injective* if $f(s) = f(s') \Rightarrow s = s'$ for $s, s' \in S$. f is *surjective* if $ran(f) = T$, and *bijective* if it is injective and surjective. For a bijective f, its *inverse function* $f^{-1} : T \to S$ such that $f^{-1}(f(s)) = s$ for $s \in S$ is well-defined and bijective, as well.

Let $S \subseteq T$ be two sets. The function $id : S \to T$ with $id(s) = s$ for all $s \in S$ is called *identity (function)* (on S). Furthermore, let S_1, S_2, and S_3 be sets, and let $f_1 : S_1 \to S_2$ and $f_2 : S_2 \to S_3$ be functions. The *composition* of f_1 and f_2 is a function $f = f_1 \circ f_2 : S_1 \to S_3$ such that $f(s_1) = f_2(f_1(s_1))$ for all $s_1 \in S_1$. Finally, let S and T be sets, let $S' \subseteq S$ be a subset of S, and let $f : S \to T$ be a function from S to T. The *restriction* of f to S' is a function $f|_{S'} : S' \to T$ with $f|_{S'}(s') = f(s')$ for all $s' \in S'$.

Let S_1 and S_2 be disjoint sets ($S_1 \cap S_2 = \emptyset$), let T_1 and T_2 be sets, and let $f_1 : S_1 \to T_1$ and $f_2 : S_2 \to T_2$ be functions. The *disjoint union* of f_1 and f_2 is a function $f_1 + f_2 : (S_1 \dot\cup S_2) \to (T_1 \cup T_2)$ that is defined by the union of their underlying relations.

Let $R \subseteq S \times S$ be a relation. R is *reflexive* if $(s, s) \in R$ for all $s \in S$, *transitive* if $(s, s') \in R \wedge (s', s'') \in R \Rightarrow (s, s'') \in R$ for all $s, s', s'' \in S$, and *symmetric* if $(s, s') \in R \Rightarrow (s', s) \in R$ for all $s, s' \in S$. R is an *equivalence relation* if R is reflexive, transitive, and symmetric.

Finally, let $R \subseteq S \times S$ be a relation. The *transitive closure* over R is a relation $R^+ \subseteq S \times S$, where $(s_1, s_n) \in R^+$ if and only if there is a sequence $s_1 \ldots s_n$ such that $(s_i, s_{i+1}) \in R$ $(1 \leq i < n,\ n > 1)$.

3.2 Graphs

A *graph* consists of typed *elements* (*nodes* and *edges*). To keep the formalization simple, we do not consider attributes. Furthermore, we refrain from introducing graph schemas (type graphs) since type consistency of graphs goes beyond the intents and the scope of our formalization.

Each edge is directed and has a unique source and a unique target. In addition to nodes, the definition permits edges as ends of edges (*higher-order edges*). We will use this feature in the representation of traces, where trace nodes are connected to edges from the source and the target graph (see Fig. 27).

Definition 1 (*Graph*) Let T_N and T_E be finite sets of *node types* and *edge types*, respectively. A *graph* over T_N and T_E is a tuple $G = (N, E, l_N, l_E, s, t)$, where

- N is a finite set of *nodes*,
- E is a finite set of *edges* ($N \cap E = \emptyset$),
- $l_N : N \to T_N$ is a *node labeling function*,
- $l_E : E \to T_E$ is an *edge labeling function*,
- $s : E \to EL$ is a *source function*, and
- $t : E \to EL$ is a *target function*.

Here, $EL = N \dot\cup E$ denotes the set of *elements* of G (nodes or edges).

Definition 2 (*Ordered graph*) Let $G = (N, E, l_N, l_E, s, t)$ be a graph over T_N and T_E, and let $EL = N \dot\cup E$ denote the set of elements of G. G is called *ordered* with respect to its edge set E if and only if there exists an ordering function $ord : EL \to \mathbb{N}$ that maps graph elements to natural numbers such that the following conditions hold:

$$\forall n \in N : ord(n) = 0 \tag{2}$$

$$\forall e \in E : ord(e) > ord(s(e)) \wedge ord(e) > ord(t(e)) \tag{3}$$

In an ordered graph, *self-referential edges* are excluded. An edge $e \in E$ is self-referential if $s(e) = e$ or $t(e) = e$. In the following, we assume that all graphs are ordered.

Example 1 The Gantt diagram on the left-hand side of Fig. 16 is represented as a graph $G = (N, E, l_N, l_E, s, t)$ as follows:

- $T_N = \{G, A, ES\}$
- $T_E = \{els, a2d, d2a\}$
- $N = \{n_1, n_2, n_3, n_4\}$
- $E = \{e_1, e_2, e_3, e_4, e_5\}$
- $l_N = \{n_1 \mapsto G, n_2 \mapsto A, n_3 \mapsto ES, n_4 \mapsto A\}$
- $l_E = \{e_1 \mapsto els, e_2 \mapsto els, e_3 \mapsto els, e_4 \mapsto a2d, e_5 \mapsto d2a\}$
- $s = \{e_1 \mapsto n_1, e_2 \mapsto n_1, e_3 \mapsto n_1, e_4 \mapsto n_2, e_5 \mapsto n_3\}$
- $t = \{e_1 \mapsto n_2, e_2 \mapsto n_3, e_3 \mapsto n_4, e_4 \mapsto n_3, e_5 \mapsto n_4\}$

G is ordered because all edges connect nodes (an ordering function that returns 0 on nodes and 1 on edges satisfies Definition 2).

Figure 23 displays G in a graphical notation, which will be used throughout the rest of this section.

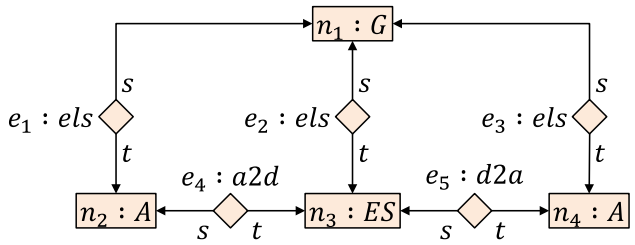

Fig. 23 Graphical representation of a Gantt diagram graph

3.3 Morphisms

Morphisms are mappings between graphs that preserve element types as well as sources and targets of edges.

Definition 3 (*Morphism*) Let $G_1 = (N_1, E_1, l_{N_1}, l_{E_1}, s_1, t_1)$ and $G_2 = (N_2, E_2, l_{N_2}, l_{E_2}, s_2, t_2)$ be graphs over T_N and T_E with element sets $EL_1 = N_1 \dot\cup E_1$ and $EL_2 = N_2 \dot\cup E_2$, respectively. A *morphism* $m : G_1 \to G_2$ is a pair of functions $m = (m_N, m_E)$ with a *node mapping function* $m_N : N_1 \to N_2$ und an *edge mapping function* $m_E : E_1 \to E_2$ such that node and edge types as well as sources and targets are preserved:

$$\forall n_1 \in N_1 : l_{N_2}(m_N(n_1)) = l_{N_1}(n_1) \tag{4}$$
$$\forall e_1 \in E_1 : l_{E_2}(m_E(e_1)) = l_{E_1}(e_1) \tag{5}$$
$$\forall e_1 \in E_1 : s_2(m_E(e_1)) = m_{EL}(s_1(e_1)) \tag{6}$$
$$\forall e_1 \in E_1 : t_2(m_E(e_1)) = m_{EL}(t_1(e_1)) \tag{7}$$

Here, $m_{EL} : EL_1 \to EL_2$ denotes the *element mapping function* $m_{EL} = m_N + m_E$.

A morphism $id = (id_N, id_E)$ with identities on nodes and edges is called an *identity morphism*.

A morphism is a *monomorphism*, *epimorphism*, or *isomorphism* if m_N and m_E are injective, surjective, or bijective, respectively.

Proposition 1 *Let $G_1 = (N_1, E_1, l_{N_1}, l_{E_1}, s_1, t_1)$ and $G_2 = (N_2, E_2, l_{N_2}, l_{E_2}, s_2, t_2)$ be graphs over T_N and T_E. Furthermore, let $m_1 : G_1 \to G_2$ and $m_2 : G_2 \to G_1$ be monomorphisms. Then both m_1 and m_2 are isomorphisms.*

Proof Since m_1 and m_2 are monomorphisms, node and edge mapping functions are injective in both directions. This implies for the cardinalities of the node sets that $|N_1| \leq |N_2|$ and $|N_2| \leq |N_1|$, resulting in $|N_1| = |N_2|$; likewise for the edge sets. Thus, additionally both m_1 and m_2 are surjective, implying that both m_1 and m_2 are isomorphisms. □

The following relations on graphs are defined with the help of morphisms.

Definition 4 (*Graph relations*) Let $G_1 = (N_1, E_1, l_{N_1}, l_{E_1}, s_1, t_1)$ and $G_2 = (N_2, E_2, l_{N_2}, l_{E_2}, s_2, t_2)$ be graphs over T_N and T_E.

- G_1 is *less or equal* to G_2 ($G_1 \lesssim G_2$) if there is a monomorphism $m : G_1 \to G_2$.
- G_1 is a *subgraph* of G_2 ($G_1 \subseteq G_2$) if and only if $id = (id_N, id_E)$ is an identity morphism from G_1 to G_2.
- G_1 is *isomorphic* to G_2 ($G_1 \simeq G_2$) if there is an isomorphism $m : G_1 \to G_2$.

Thus, $G_1 \subseteq G_2$ implies $G_1 \lesssim G_2$. Furthermore, $\lesssim$ and $\subseteq$ are reflexive and transitive. Finally, $\simeq$ is an equivalence relation. The proofs are straightforward.

Definition 5 (*Morphism operators*) Let $G_i = (N_i, E_i, l_{N_i}, l_{E_i}, s_i, t_i)$ $(1 \leq i \leq 3)$ be graphs over T_N and T_E.

- Let $m_1 : G_1 \to G_2$ and $m_2 : G_2 \to G_3$ be morphisms, where $m_1 = (m_{N_1}, m_{E_1})$ and $m_2 = (m_{N_2}, m_{E_2})$, respectively. The *composition* of m_1 and m_2 is a morphism $m = m_1 \circ m_2 : G_1 \to G_3$ with $m = (m_{N_1} \circ m_{N_2}, m_{E_1} \circ m_{E_2})$.
- Let $m : G_1 \to G_2$ be an isomorphism $m = (m_N, m_E)$. The *inverse* of m is an isomorphism $m^{-1} : G_2 \to G_1$ with $m^{-1} = ({m_N}^{-1}, {m_E}^{-1})$.
- Let $m : G_1 \to G_2$ be a morphism with $m = (m_N, m_E)$, and let $G_3 \subseteq G_1$ be a subgraph of G_1. The *restriction* of m onto G_3 is a morphism $m|_{G_3} : G_3 \to G_2$ such that $m|_{G_3} = (m_N|_{N_3}, m_E|_{E_3})$.
- Let $m : G_1 \to G_2$ be a morphism with $m = (m_N, m_E)$. The *range* of m ($ran(m)$) is a subgraph $G_3 \subseteq G_2$ with node set N_3 and edge set E_3 such that $N_3 = ran(m_N)$ and $E_3 = ran(m_E)$.

It may be checked easily that the definitions are sound (e.g., the inverse of an isomorphism is an isomorphism).

Proposition 2 *The following statements hold for morphism operators (G_1, G_2, and G_3 denote graphs over T_N and T_E):*

1. *Let $m_1 : G_1 \to G_2$ and $m_2 : G_2 \to G_3$ be monomorphisms. Then $m_1 \circ m_2 : G_1 \to G_3$ is a monomorphism, as well.*
2. *Let $m : G_1 \to G_2$ be a morphism with $m = (m_N, m_E)$. Then the morphism $m' : G_1 \to ran(m)$ with $m' = (m'_N, m'_E)$ and $m'_N(n) = m_N(n), m'_E(e) = m_E(e)$ for all $n \in N_1$, $e \in E_1$ (where N_1 and E_1 denote the node set and the edge set of G_1, respectively) is an epimorphism.*

Proof Follows immediately from the definitions. □

3.4 Rules and derivations

This section defines rules and derivations conforming to the computational model that is introduced in Sect. 2.5.1.

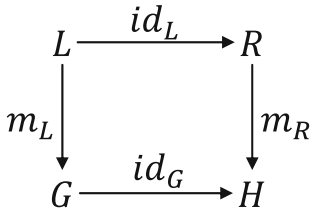

Fig. 24 Application of a rule

Definition 6 (*Rule and rule set*) A *rule (production)* over T_N and T_E is a pair $p = (L, R)$, where L and R are graphs over T_N and T_E such that $L \subseteq R$. L and R are called the *left-hand side* and the *right-hand side*, respectively. A *rule set* P is a finite set of rules $p \in P$.

Definition 7 (*Match*) Let $p = (L, R)$ and G be a rule and a graph over T_N and T_E, respectively. A *match* for p in G is a monomorphism $m_L : L \to G$.

The graph G to which the rule is applied is called *host graph*.

Definition 8 (*Direct derivation*) Let G and H be graphs over T_N and T_E. Let $p = (L, R)$ be a rule over T_N and T_E and $m_L : L \to G$ be a match for p in G. H is *directly derivable from G via p* if H satisfies the following properties:

- A monomorphism $m_R : R \to H$ exists such that the diagram of Fig. 24 commutes ($id_L : L \to R$ and $id_G : G \to H$ denote identity morphisms): $id_L \circ m_R = m_L \circ id_G$.
- For each graph H' with the same properties, $H \lesssim H'$.

We write $G \overset{p}{\to} H$ or $G \overset{p,m_L}{\longrightarrow} H$ if the match m_L is important. An application of a rule to a graph is also called a *derivation step*.

Finally, let P be a rule set. H is *directly derivable from G via P* ($G \overset{P}{\to} H$) if there is a rule $p \in P$ such that $G \overset{p}{\to} H$ holds.

Thus, when a rule is applied to a match, the elements of the right-hand side that are not part of the left-hand side are added to the host graph.

Rules satisfy the properties of the computational model introduced in Sect. 2.5.1:

Proposition 3 *Let G, p, and m_L be a graph, a rule, and a match for p in G, respectively. Then, the following* properties *hold with respect to the application of p:*

1. *p is* functional *(Property 4), i.e., once a match m_L for p is fixed, the resulting graph H is unique up to isomorphism.*
2. *p is* monotonic *(Property 5), i.e., p only adds nodes or edges.*
3. *p is* local *(Property 6), i.e., the condition under which p may be applied as well as its effect depends only on the match m_L.*

Proof The statements of the proposition are proved as follows:

1. Let H and H' be derivable from G via p and match m_L. Then, $H \lesssim H'$ and $H' \lesssim H$. According to Proposition 1, $H \simeq H'$.
2. Since id_G is an identity morphism, all elements of G are preserved.
3. According to Definition 7, a rule p is applicable if and only if a match $m_L : L \to G$ may be found. According to Definition 8, the effect of applying a rule depends only on the match (elements to be inserted are connected only to elements from $m_L(L)$).

□

Definition 9 (*Derivation*) Let G and H be graphs, and let P be a rule set. H is *derivable from G via P* ($G \overset{P^*}{\to} H$ or simply $G \overset{*}{\to} H$) if and only if there is a natural number $n \geq 0$, a sequence of graphs $G_0 \ldots G_n$, a sequence of rules $p_0 \ldots p_{n-1}$ from P, and a sequence of matches $m_0 \ldots m_{n-1}$ such that the following conditions hold:

- $G = G_0$, $H = G_n$
- $G_i \overset{p_i,m_i}{\longrightarrow} G_{i+1}$ for $0 \leq i < n$
- No rule may be applied more than once to the same match: $i \neq j \Rightarrow (p_i, m_i) \neq (p_j, m_j)$ for $0 \leq i, j < n$

A sequence of derivation steps satisfying the conditions stated above is called a *derivation*. A derivation $G \overset{*}{\to} H$ is *complete* if it cannot be extended any more.

A complete derivation is a *rule-based transformation* in the sense of Property 3: all rules are applied to all matches exactly once.

3.5 Properties of derivations

Below, we prove some properties of derivations that are used later to conduct the proof of commutativity (Sect. 3.9).

3.5.1 Monotonicity

It follows by induction from Proposition 3 that the monotonicity of rules carries over to derivations:

Proposition 4 *Let P be a set of rules, and let $G_0 \overset{P}{\to} G_1 \overset{P}{\to} \ldots G_{n-1} \overset{P}{\to} G_n$ denote a derivation via P ($n > 0$). Then $G_i \subseteq G_{i+1}$ for each $0 \leq i < n$.*

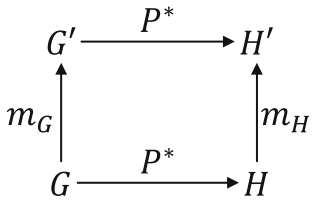

Fig. 25 Monomorphism preservation

The following proposition establishes a monotonicity result for derivations on graphs G and G' such that $G \lesssim G'$:

Proposition 5 *Let P be a set of rules. Let G and G' be graphs such that $G \lesssim G'$. Furthermore, let $G \overset{P^*}{\to} H$ be a derivation and $G' \overset{P^*}{\to} H'$ be a complete derivation. Then $H \lesssim H'$.*

Proof Since $G \lesssim G'$, there is a monomorphism $m_G : G \to G'$. We will prove that there is a monomorphism $m_H : H \to H'$ that completes the diagram of Fig. 25, where $G \overset{P^*}{\to} H$, $G' \overset{P^*}{\to} H'$, and m_G are given.

For $G \overset{P^*}{\to} H$, let $G_0 \dots G_n$, $p_0 \dots p_{n-1}$, and $m_0 \dots m_{n-1}$ be sequences of graphs, rules, and matches satisfying the conditions from Definition 9 ($n \geq 0$). Likewise, for $G' \overset{P^*}{\to} H'$ the corresponding sequences are denoted by $G'_0 \dots G'_{n'}$, $p'_0 \dots p'_{n'-1}$, and $m'_0 \dots m'_{n'-1}$, respectively ($n' \geq 0$).

We construct a sequence of monomorphisms $m_{G_i} : G_i \to H' (0 \leq i \leq n)$ such that $m_H = m_{G_n} : H \to H'$ is a monomorphism from H to H'. Furthermore, we construct a sequence of injective index functions $f_i : [0..i-1] \to [0..n'-1]$ $(0 \leq i \leq n)$, where f_n maps each derivation step (p_i, m_i) to a corresponding derivation step (p'_j, m'_j) such that both derivation steps apply the same rule $p'_j = p_i$ to corresponding matches.

m_{G_i} and f_i are constructed inductively. For $i = 0$ (start of induction), we set $m_{G_0} = m_G$ (which is a monomorphism by assumption) and $f_0 = \emptyset$ (a function that is defined on an empty interval because no derivation step has been mapped yet).

For the induction step $i \to i+1$, let assume that m_{G_i} and f_i satisfy the properties stated above $(0 \leq i < n)$. Let $G_i \overset{p_i, m_i}{\longrightarrow} G_{i+1}$ denote the next derivation step to be processed, where $p_i = (L_i, R_i)$ and $m_i : L_i \to G_i$ is a monomorphism. Since $m_{G_i} : G_i \to H'$ is a monomorphism by the inductive assumption, the composition $m_i \circ m_{G_i} : L_i \to H'$ is a monomorphism, as well (Proposition 2), and thus, a match for p_i in H'. Since all rules are applied to all matches (complete derivation), there is a derivation step (p'_j, m'_j) such that $p'_j = p_i$ and $m'_j = m_i \circ m_{G_i}$. Note that p_i is applicable to m'_j because rules are local.

j is not a member of the range of f_i: $j \notin ran(f_i)$. This is proved by contradiction: Let us assume that there is an index $0 \leq k < i$ such that $f_i(k) = j$. For the derivation step (p_k, m_k), $p_k = p_i$ holds. Furthermore, $m_k \circ m_{G_i} = m_i \circ m_{G_i}$. Since m_{G_i} is a monomorphism and thus injective, this implies $m_k = m_i$. Altogether, $(p_k, m_k) = (p_i, m_i)$ for $k \neq i$. This is a contradiction to the uniqueness of derivation steps in derivations (each rule is applied to each match only once).

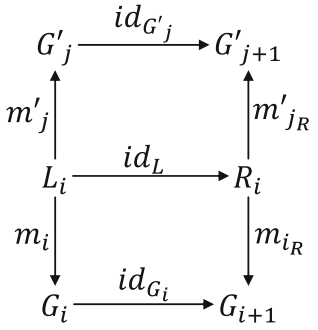

Fig. 26 Morphisms for rule applications in G and G'

Thus, f_{i+1} is constructed from f_i as follows:

$$f_{i+1} = f_i \cup \{i \mapsto j\} \tag{8}$$

$dom(f_{i+1}) = [0..i]$. Furthermore, f_{i+1} is injective. Finally, $p'_j = p_i$, and $m'_j = m_i \circ m_{G_i}$ is a match for p_i in G'_j. Altogether, the inductive invariant regarding the index function is established.

To extend m_{G_i} into $m_{G_{i+1}}$, the graph elements created by the derivation step (p_i, m_i) have to be mapped to H'. By applying Definition 8, we obtain the diagram of monomorphisms shown in Fig. 26. Since m_{i_R} is a monomorphism, $m_{i_R} : R_i \to ran(m_{i_R})$ is an isomorphism (Proposition 2). Therefore, the inverse morphism $m_{i_R}^{-1} : ran(m_{i_R}) \to R_i$ exists and is an isomorphism, as well. Furthermore, $m_{i_R}^{-1} \circ m'_{j_R} : ran(m_{i_R}) \to G'_{j+1}$ is a monomorphism that maps elements matched or created by (p_i, m_i) to elements matched or created by (p'_j, m'_j) (where $p_i = p'_j$).

Let N_i and E_i denote the sets of new graph elements created by the derivation step (p_i, m_i). Let $m_i = (m_{i_N}, m_{i_E})$, $m_{i_R} = (m_{i_{R_N}}, m_{i_{R_E}})$. N_i and E_i are computed as follows:

$$N_i = ran(m_{i_{R_N}}) \setminus ran(m_{i_N}) \tag{9}$$

$$E_i = ran(m_{i_{R_E}}) \setminus ran(m_{i_E}) \tag{10}$$

Let $m_{G_i} = (m_{G_{i_N}}, m_{G_{i_E}})$ and $m_{G_{i+1}} = (m_{G_{i+1_N}}, m_{G_{i+1_E}})$. Furthermore, let $m_R = m_{i_R}^{-1} \circ m'_{j_R}$, where $m_R = (m_{R_N}, m_{R_E})$, denote the monomorphism mapping the right-hand sides of the rule applications in G_{i+1} and G'_{j+1} onto each other. The new node and edge mapping functions are calculated as follows:

$$m_{G_{i+1_N}} = m_{G_{i_N}} \cup \{n \mapsto m_{R_N}(n) \mid n \in N_i\} \tag{11}$$

$$m_{G_{i+1_E}} = m_{G_{i_E}} \cup \{e \mapsto m_{R_E}(e) \mid e \in E_i\} \quad (12)$$

Since the new nodes and edges in G_{i+1} are mapped monomorphically onto new nodes and edges in ${G'}_{j+1}$, $m_{G_{i+1}}$ is a monomorphism from G_{i+1} to H'. This completes the proof of the inductive step.

Altogether, $m_H = m_{G_n}$ is a monomorphism from $H = G_n$ to H', and $f_n : [0..n-1]$ injectively maps derivation steps on G to derivation steps on G'. □

It should be noted that the proof exploits the assumption that $G' \overset{P^*}{\to} H'$ is a complete derivation. Otherwise, the existence of a corresponding derivation step in $G' \overset{P^*}{\to} H'$ cannot be deduced.

3.5.2 Functional behavior

Complete derivations exhibit *functional behavior*: the result of a derivation is determined uniquely up to isomorphism.

Proposition 6 *Let P be a set of rules, and let G be a graph. Furthermore, let $G \overset{P^*}{\to} H_1$ and $G \overset{P^*}{\to} H_2$ be complete derivations. Then H_1 and H_2 are isomorphic: $H_1 \simeq H_2$.*

Proof We apply Proposition 5 twice. The identity on G is a monomorphism of G into itself. For $H = H_1$ and $H' = H_2$, we obtain a monomorphism $m_{H_1} : H_1 \to H_2$; switching the order yields a monomorphism $m_{H_2} : H_2 \to H_1$. According to Proposition 1, both m_{H_1} and m_{H_2} are isomorphisms. Thus, $H_1 \simeq H_2$. □

It should be noted that both derivations are required to be complete. Otherwise, Proposition 5 cannot be applied in both directions.

3.5.3 Termination

In general, the computational model does not guarantee termination: Rules may constantly generate new matches, resulting in an *infinite sequence of derivation steps*. However, if there is a complete derivation starting from some graph G', any sequence of derivation steps starting on a graph $G \lesssim G'$ is finite.

Proposition 7 *Let P be a set of rules. Let $G \lesssim G'$ be graphs. Furthermore, let $G \overset{P^*}{\to} H$ be a derivation and $G' \overset{P^*}{\to} H'$ be a complete derivation. Let n and n' denote the numbers of derivation steps in $G \overset{P^*}{\to} H$ and $G' \overset{P^*}{\to} H'$, respectively. Then $n \leq n'$.*

Proof The bounded length of derivations starting from G follows immediately from the construction performed in the proof of Proposition 5, where derivation steps in $G \overset{P^*}{\to} H$ are mapped injectively to derivation steps in $G' \overset{P^*}{\to} H'$. □

In particular, if we select $G = G'$, we find that the existence of a complete derivation starting from G implies that any sequence of derivation steps starting from G will terminate. Furthermore, as shown above, all complete derivations yield the same result (up to isomorphism).

3.6 Source-to-target transformations

As introduced so far, rules constitute *in-place transformations*: a single graph is transformed by applying rules. In the following, we simulate *out-place transformations* by decomposing graphs into multiple subgraphs (a *source graph*, a *trace graph*, and a *target graph*). Rules are constrained such that they do not modify the source graph. A *batch transformation* is defined as a complete derivation that applies *trace-generating source-to-target rules*, starting with a graph whose trace and target subgraphs are initially empty. Altogether, the definitions from this section contribute all properties of the computational model (Sect. 2.5.1) that have been missing so far because they do not hold for the computational base model.

Definition 10 (*Source-to-target graph*) A *source-to-target graph (STT graph)* is a graph G that is composed of three mutually disjoint subgraphs and edge sets connecting these subgraphs (and no further nodes and edges):

- A *source graph* $G_S \subseteq G$ that is typed over node types T_{N_S} and edge types T_{E_S}.
- A *target graph* $G_T \subseteq G$ that is typed over node types T_{N_T} and edge types T_{E_T}.
- A *trace graph* $G_{TR} \subseteq G$ that is typed over node types $T_{N_{TR}}$ and edge types $T_{E_{TR}}$. Trace nodes are typed over rule identifiers, i.e., $T_{N_{TR}} = ID_P$ for some rule set P. The edge type set $T_{E_{TR}}$ contains a single edge type: $T_{E_{TR}} = \{use\}$.
- *Trace-to-source edges* of type src from nodes of the trace graph to elements of the source graph.
- *Trace-to-target edges* of type ctx or trg from nodes of the trace graph to elements of the target graph.

The notation $G = G_S \leftarrow G_{TR} \rightarrow G_T$ is used to indicate that G is an STT graph with the components listed above.

To simplify notation, node and edge types are not mentioned explicitly below. Furthermore, we refrain from expanding graphs explicitly into components. Rather, we follow the convention that indices denote the respective underlying graph. For example, N_{G_S} denotes the node set of G_S.

Example 2 Figure 27, a refinement of parts of Fig. 16, shows an example of an STT graph. The graph shows a state that is reached after two rules have been applied: the first rule

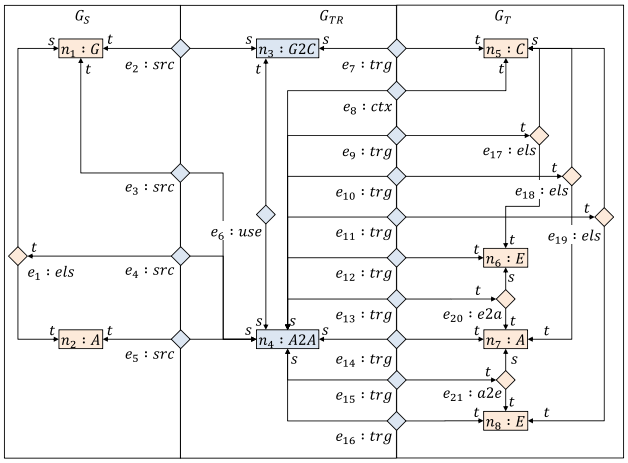

Fig. 27 STT graph

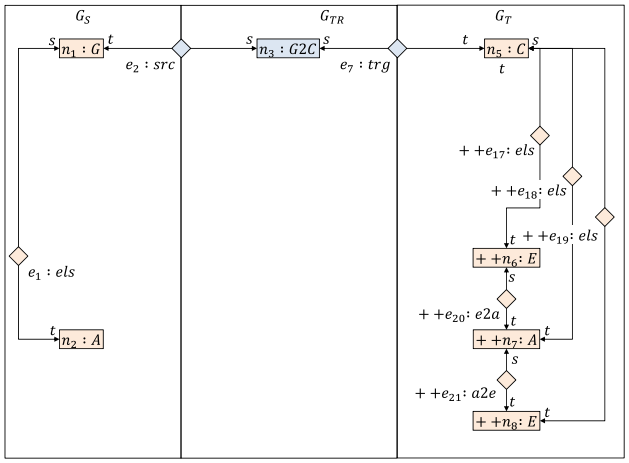

Fig. 28 STT rule for mapping activities

maps the root of the Gantt diagram to the root of the CPM network; the second rule maps an activity in a Gantt diagram to an activity of the CPM network. The trace-to-source and trace-to-target edges, although connecting elements of different subgraphs, are considered as part of the trace and are thus displayed in blue color. The *use* edge e_6 from n_4 to n_3 indicates that the rule for mapping the activity uses the result of the rule for mapping the diagram.

A *source-to-target rule* is a rule on STT graphs that is used to extend the target graph. The rule may access the trace, but it does not extend it:

Definition 11 (*Source-to-target rule*) Let $p = (L, R)$ be a rule, where $L = L_S \leftarrow L_{TR} \rightarrow L_T$ and $R = R_S \leftarrow R_{TR} \rightarrow R_T$ are STT graphs. p is a *source-to-target rule (STT rule)* if all elements that are not contained in L are members of R_T:

$$\forall el \in EL_R : el \notin EL_L \Rightarrow el \in EL_{R_T} \tag{13}$$

Furthermore, each element in L_T must have exactly one incoming trg edge:

$$\forall el \in EL_{L_T} : |\{e \in E_L | t_L(e) = el \wedge l_{E_L}(e) = trg\}| = 1 \tag{14}$$

The existence of an incoming trg edge for elements of the left-hand side of the target graph (Eq. 14) ensures completeness of dependency information (see Proposition 8).

Example 3 Figure 28 shows an STT rule for mapping activities that corresponds to the informal rule of Fig. 20. Again, ++ markers indicate created nodes and edges.

In order to capture trace information, user-defined STT rules are transformed automatically into *trace-generating STT rules*, which are used for actual execution. Tools such as ATL/EMFTVM and medini QVT are based on a similar approach. Thus, the user may conveniently define STT rules accessing trace information but is not required to specify the generation of trace information. Rather, the trace is built up automatically during execution.

Definition 12 (*Trace-generating STT rule*) Let $p = (L, R)$ be an STT rule, where $L = L_S \leftarrow L_{TR} \rightarrow L_T$ and $R = R_S \leftarrow R_{TR} \rightarrow R_T$ are STT graphs. The *trace-generating STT rule* $p' = (L', R')$ with $L' = L'_S \leftarrow L'_{TR} \rightarrow L'_T$ and $R' = R'_S \leftarrow R'_{TR} \rightarrow R'_T$ is constructed from p by executing the following sequence of steps:

1. Initialize p' with p: $p' := p$. Subsequently, extend the right-hand side R' as described below.
2. Add a single trace node n' to the trace graph that is typed by the identifier id_p of rule p:

$$l'_N(n') = id_p \tag{15}$$

3. For each old trace node n from p, create a trace edge e' of type use from the new to the old trace node:

$$l'_E(e') = use \wedge s'(e') = n' \wedge t'(e') = n \tag{16}$$

4. For each source element (node or edge) el from p, create a trace edge e' of type src from the new trace node to the source element:

$$l'_E(e') = src \wedge s'(e') = n' \wedge t'(e') = el \tag{17}$$

5. For each old target element el from p, create a trace edge e' of type ctx from the new trace node to the old target element:

$$l'_E(e') = ctx \wedge s'(e') = n' \wedge t'(e') = el \tag{18}$$

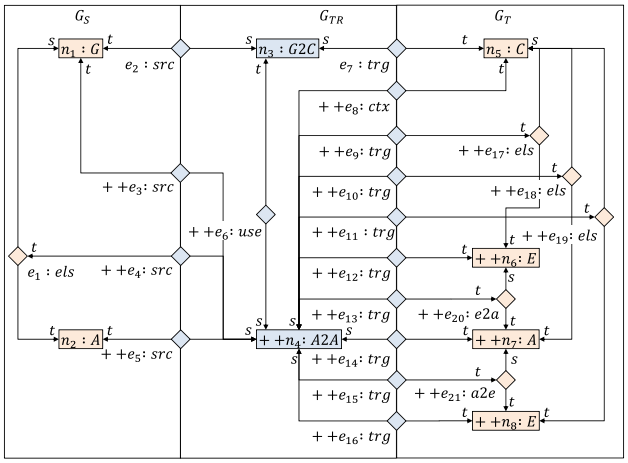

Fig. 29 Trace-generating STT rule for mapping activities

6. For each new target element el from p, create a trace edge e' of type trg from the new trace node to the new target element:

$$l'_E(e') = trg \wedge s'(e') = n' \wedge t'(e') = el \tag{19}$$

Example 4 Figure 29 shows the trace-generating STT rule that is obtained from the STT rule for mapping activities (Fig. 28).

Based on the construction of STT rules, let us classify the traces of our computational model according to the taxonomy displayed in Fig. 13:

- Traces are *rule-based*; thus, each trace node is typed by the rule whose application is represented by the trace node.
- Traces are *complete* inasmuch as they capture all source, context, and target elements (Property 7).
- Traces are *fine-grained* since they include not only nodes but also edges (Property 8).

Definition 13 (*STT derivation*) Let $G = G_S \leftarrow G_{TR} \rightarrow G_T$ and $H = H_S \leftarrow H_{TR} \rightarrow H_T$ be STT graphs. Furthermore, let P be a set of trace-generating STT rules. *H is STT-derivable from G* if $G \overset{P^*}{\rightarrow} H$ holds.

Thus, an STT derivation is a derivation (Definition 9) with trace-generating STT rules. The notion of *completeness* carries over, as well. Since STT rules do not modify the source graph, $G_S = H_S$ always holds.

The proposition below collects various facts that are relevant for the propagation algorithm. Note that we use the symbol $\emptyset$ to denote an empty graph (with empty element set). Furthermore, we identify edge types t_E with binary relations; i.e., we write $(el_1, el_2) \in t_E$ if there is an edge of type t_E from el_1 to el_2. For $(el_1, el_2) \in t_E$, we use arrow notation $el_1 \overset{t_E}{\rightarrow} el_2$.

Proposition 8 *Let P be a set of trace-generating STT rules. Let $G = G_S \leftarrow \emptyset \rightarrow \emptyset$ and $H = H_S \leftarrow H_{TR} \rightarrow H_T$ be STT graphs such that H is STT-derivable from G via P. The following properties hold for H:*

1. *Each element of H_T has exactly one incoming trg edge:*

$$\forall el \in EL_{H_T} : |\{e \in E_H | t_H(e) = el \wedge l_{E_H}(e) = trg\}| = 1 \tag{20}$$

2. *H_{TR} is acylic with respect to use edges: let $use^+ \subseteq N_{H_{TR}} \times N_{H_{TR}}$ denote the transitive closure over use edges. No trace node uses itself:*

$$\forall n_{H_{TR}} \in N_{H_{TR}} : \neg (n_{H_{TR}} \overset{use^+}{\longrightarrow} n_{H_{TR}}) \tag{21}$$

3. *Each create/use dependency between rule applications is explicit: when a trace node has an outgoing ctx edge to a target element, it also has an outgoing use edge to another trace node that has an outgoing trg edge to this target element:*

$$\begin{aligned} &\forall n_{H_{TR}} \in N_{H_{TR}},\ el_{H_T} \in EL_{H_T} : n_{H_{TR}} \overset{ctx}{\longrightarrow} el_{H_T} \Rightarrow \\ &\exists n'_{H_{TR}} \in N_{H_{TR}} : n_{H_{TR}} \overset{use}{\longrightarrow} n'_{H_{TR}} \wedge n'_{H_{TR}} \overset{trg}{\longrightarrow} el_{H_T} \end{aligned} \tag{22}$$

Proof For $G \overset{P^*}{\rightarrow} H$, let $G_0 \ldots G_n$ and $p_0 \ldots p_{n-1}$ be sequences of graphs and rules such that $G = G_0$, $H = G_n$, and $G_i \overset{p_i}{\rightarrow} G_{i+1}$ $(0 \leq i < n)$. We prove by induction over i that each G_i satisfies the properties stated in the proposition.

Start of induction ($i = 0$): since $G_0 = G$, the trace graph G_{TR_0} and the target graph G_{T_0} are empty. Thus, all properties are satisfied.

Induction step ($i \rightarrow i + 1$): let all properties hold for G_i. Let $p_i = (L_i, R_i)$ with $L_i = L_{S_i} \leftarrow L_{TR_i} \rightarrow L_{T_i}$ and $R_i = R_{S_i} \leftarrow R_{TR_i} \rightarrow R_{T_i}$ denote a trace-generating STT rule. Applying p_i maintains the inductive invariant, implying that all properties hold for G_{i+1}:

1. By construction of p_i, all new elements receive exactly one incoming trg edge (Eq. 19). No further trg edges are created. Altogether, all (old and new) elements of G_{i+1} have exactly one incoming trg edge.
2. By construction of p_i, each created use edge emanates from the new trace node $n_{R_{TR_i}} \in (N_{R_{TR_i}} \setminus N_{L_{TR_i}})$ created by p_i (Eq. 16). No further use edges are created. Since the trace node is created in the same derivation step as

its outgoing *use* edges (Definition 12), the trace graph $G_{TR_{i+1}}$ remains acyclic.

3. By the inductive assumption, Eq. 22 holds for all *ctx* edges in G_i. By applying p_i, new *ctx* edges are created to elements from G_{T_i} (Eq. 18). According to Definition 11, in p_i each element from L_i must have an incoming *trg* edge (Eq. 14), which emanates from some trace node. Furthermore, a *use* edge from the new trace node to the old trace node is created by applying p_i (Eq. 16). Altogether, the conclusion in Eq. 22 holds for all new *ctx* edges. Using the inductive assumption, Eq. 22 holds for G_{i+1}.

□

Definition 14 (*Single-variant derivation*) Let S and T be a source graph and a target graph, respectively. Furthermore, let P be a set of trace-generating STT rules. T *is single-variant derivable from* S ($S \overset{P^*}{\rightarrow} T$) if there are STT graphs $G = G_S \leftarrow G_{TR} \rightarrow G_T$ and $H = H_S \leftarrow H_{TR} \rightarrow H_T$ such that the following conditions hold:

$$G_S = S \wedge G_{TR} = \emptyset \wedge G_T = \emptyset \tag{23}$$

$$G \overset{P^*}{\rightarrow} H \tag{24}$$

$$H_T = T \tag{25}$$

$G \overset{P^*}{\rightarrow} H$ is required to be a complete STT derivation.

Thus, Definition 14 defines a relation between source and target graphs with the help of a relation between STT graphs (Definition 13). In this way, *out-place batch transformations* are formalized (Properties 1, 2, respectively).

Altogether, the computational model, defined in Sects. 3.4 and 3.6, satisfies all properties listed in Sect. 2.5.1.

Example 5 In Fig. 27, the target graph G_T has been derived from the source graph G_S according to Definition 14.

3.7 Variability

In PLE, *feature models* [22] are frequently used to express the variability of a product line. In this paper, we do not assume a specific approach to variability modeling but adopt the terminology from feature models. We assume that the common and discriminating properties of members of a product line are defined by Boolean *features* from a set $F = \{f_1, \ldots, f_k\} (k \geq 1)$.

Definition 15 (*Feature expression*) Let F be a set of features. A *feature expression* over F is an expression in propositional logic over features from F. The set of all feature expressions over F is denoted by E_F; we write e_F for a member of E_F.

Feature configurations are used to define product variants. A feature configuration defines a product variant by determining the features to be included into that variant. Feature configurations may be defined as specific kinds of feature expressions:

Definition 16 (*Feature configuration*) Let $F = \{f_1, \ldots, f_k\}$ be a set of features. A *feature configuration* over F is a conjunction of bindings $c_F = b_1 \wedge \ldots \wedge b_k$, where $b_i \in \{f_i, \neg f_i\}$. The set of all feature configurations is denoted by C_F.

Thus, a positive binding means that the respective feature is included, and a negative binding means that it is omitted from the specified product variant. Feature configurations are *fully bound*; each feature is bound to a truth value.

A *variability model* determines which feature configurations are *valid*. In the following, we assume valid feature configurations but we refrain from formally defining variability models (which is not needed for our purposes).

In annotative approaches, variability is expressed by attaching *feature expressions* to model elements. We assume an *evaluation function* that takes a feature configuration and a feature expression and returns a Boolean result:

Definition 17 (*Evaluation function*) Let F be a set of features, C_F a set of feature configurations over F, and E_F a set of feature expressions over F. An *evaluation function* over F is a function $v_F : E_F \times C_F \rightarrow \mathbb{B}$, where $\mathbb{B} = \{true, false\}$ denotes the set of Boolean values. v_F returns the value $true$ if and only if e_F may be deduced from c_F:

$$c_F \vdash e_F \tag{26}$$

For example, for $e_F = f_1 \vee f_2$ the call $v_F(e_F, c_F)$ returns $true$ for $c_F = f_1 \wedge \neg f_2$ and $false$ for $c_F = \neg f_1 \wedge \neg f_2$.

A *multi-variant graph* is a graph whose elements are decorated with feature expressions:

Definition 18 (*Multi-variant graph*) Let F be a set of features. A *multi-variant graph* is a pair $G_F = (G, a_F)$, where $G = (N, E, l_N, l_E, s, t)$ is a graph, $EL = N \dot{\cup} E$, and $a_F : EL \rightarrow E_F$ is an *annotation function* that decorates each element with a feature expression. For each edge $e \in E$, a_F must satisfy the following constraint (*referential consistency*):

$$a_F(e) \Rightarrow a_F(s(e)) \wedge a_F(t(e)) \tag{27}$$

The constraint for the annotation function ensures that a single-variant graph obtained by filtering a multi-variant graph returns a valid graph, without dangling edges.

Example 6 Figure 30 shows a multi-variant diagram graph. As in Fig. 16, we assume a mandatory root feature R and two optional subfeatures F and G.

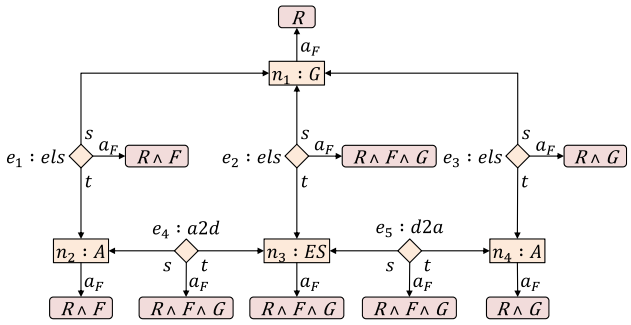

Fig. 30 Example of a multi-variant diagram graph

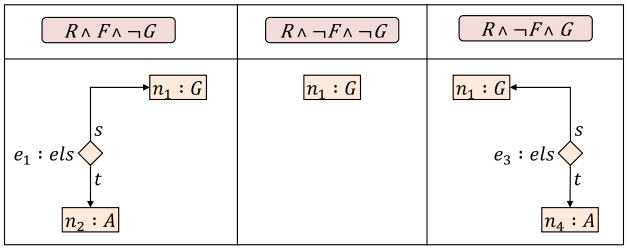

Fig. 31 Variants of diagram graphs

A *filter* is a function that reduces the underlying single-variant graph G of a multi-variant graph G_F to those nodes and edges whose feature expressions evaluate to $true$ under the given feature configuration.

Definition 19 (*Filter*) Let F be a set of features and C_F be the corresponding set of feature configurations. Let $\mathcal{G}_F$ and $\mathcal{G}$ denote the sets of all multi-variant graphs and single-variant graphs, respectively. A *(graph) filter* is a function $f_F : \mathcal{G}_F \times C_F \rightarrow \mathcal{G}$ that is defined as follows: for some multi-variant graph $G_F = (G_1, a_{F_1})$ and some feature configuration c_F, the filter returns a graph $G_2 \subseteq G_1$ whose node and edge sets are defined as follows:

$$N_2 = \{n_1 \in N_1 \mid v_F(a_{F_1}(n_1), c_F) = true\} \tag{28}$$

$$E_2 = \{e_1 \in E_1 \mid v_F(a_{F_1}(e_1), c_F) = true\} \tag{29}$$

Example 7 Assuming a mandatory root feature R and two optional subfeatures F and G, the multi-variant graph from Fig. 30 may be configured into four variants. The variant for the feature configuration $R \wedge F \wedge G$ was already shown in Fig. 23. The remaining variants are displayed in a tabular form in Fig. 31, where the upper and the lower row show feature configurations and corresponding graph variants, respectively.

3.8 Propagation algorithm

Algorithm 1—written in pseudo code—propagates annotations from the source to the target model by processing each trace node in turn. For each rule application, each target element is decorated with the conjunction of the annotations of all source and context elements (Line 19). The generated expression is simplified (Line 21) to reduce the size of the annotations (otherwise, it would contain repeated subexpressions if multiple elements are annotated with the same expression).

The function SELECT called in Line 13 ensures that each rule application is processed only after all rule applications on which it depends (i.e., the trace node to be selected must not have outgoing *use* edges to other trace nodes in the working set). SELECT implements a topological sort that guarantees that the annotations of context elements have already been assigned when they are used to annotate target elements.

For the sake of a uniform representation throughout the whole formalization section, Algorithm 1 is described in terms of the graph-based computational model introduced earlier in this section. However, it should be noted that trace-based propagation was actually implemented against the abstract *trace metamodel* from Fig. 14. With the information stored in STT graphs, it is straightforward to implement this abstract interface. The sequential order of trace elements is implemented by a topological sort; the references to source, context, and target elements are implemented as shown in Lines 16–18.

The following proposition states that trace-based propagation produces a consistently annotated target graph:

Proposition 9 *Let $H = H_S \leftarrow H_{TR} \rightarrow H_T$ be an STT graph, derived from the STT graph $H_S \leftarrow \emptyset \rightarrow \emptyset$. Let $a_{F_{H_S}} : EL_{H_S} \rightarrow E_F$ be the annotation function for the source graph H_S. Finally, let $a_{F_{H_T}} : EL_{H_T} \rightarrow E_F$ be the annotation function produced by Algorithm 1. Then, $H_{T_F} = (H_T, a_{F_{H_T}})$ is a multi-variant target graph.*

Proof The proof is composed of three parts:

1. The annotation function constructed by trace-based propagation is *total*: all derivation steps are processed, each target element of each derivation step is annotated, and each element of H_T has exactly one incoming trg edge (Proposition 8, Eq. 20).
2. The annotation function is *well-defined*: since the trace graph is acylic (Proposition 8, Eq. 21), a topological sort may be performed (by the SELECT function) to ensure that each trace node is processed only after all its used trace nodes. Furthermore, each context element of a trace node has an incoming trg edge from some used trace node (Proposition 8, Eq. 22). Thus, the annotation of a context element is accessed only after it has been assigned when the used trace node has been processed. Therefore, the feature expression in Line 19 of Algorithm 1 is well-defined.

Algorithm 1 Propagation of annotations

```
1: procedure PROPAGATE(H, a_{F_{H_S}}, a_{F_{H_T}})
2:     in H = H_S ← H_TR → H_T                          ▷ STT graph, derived from H_S ← ∅ → ∅
3:     in a_{F_{H_S}} : EL_{H_S} → E_F                  ▷ Annotation function for the source graph H_S
4:     out a_{F_{H_T}} : EL_{H_T} → E_F                 ▷ Annotation function for the target graph H_T
5:     var W_{H_TR} ⊆ N_{H_TR}                          ▷ Working set of trace nodes
6:     var n_{H_TR} ∈ N_{H_TR}                          ▷ The current trace node to be processed
7:     var SE ⊆ EL_{H_S}, CE ⊆ EL_{H_T}, TE ⊆ EL_{H_T}  ▷ Source, context, and target elements
8:     var e_F ∈ E_F                                    ▷ Feature expression to be assigned to target elements
9:     var te ∈ EL_{H_T}                                ▷ The target element to be annotated
10:
11:    W_{H_TR} := N_{H_TR}                             ▷ Initialize working set of trace nodes
12:    while W_{H_TR} ≠ ∅ do                            ▷ Working set is not empty
13:        n_{H_TR} :=SELECT(W_{H_TR})                  ▷ Select trace node in topological order
14:        W_{H_TR} := W_{H_TR} \ {n_{H_TR}}            ▷ Remove trace node from working set
15:                                                     ▷ Determine source, context, and target elements (defined using arrow notation for edges):
16:        SE := {se ∈ EL_{H_S} | n_{H_TR} --src--> se}
17:        CE := {ce ∈ EL_{H_T} | n_{H_TR} --ctx--> ce}
18:        TE := {te ∈ EL_{H_T} | n_{H_TR} --trg--> te}
19:        e_F := ⋀{a_{F_S}(se) | se ∈ SE} ∧ ⋀{a_{F_T}(ce) | ce ∈ CE}
20:                                                     ▷ The feature expression is a conjunction of source and context element expressions
21:        e_F := SIMPLIFY(e_F)                         ▷ Simplify resulting expression
22:        for te ∈ TE do                               ▷ Process all target elements
23:            a_{F_{H_T}}(te) := e_F                   ▷ Annotate the target element
24:        end for
25:    end while
26: end procedure
```

3. Finally, the annotations are *referentially consistent*, i.e., the feature expression attached to an edge implies the feature expressions of its ends (Definition 18):

$$\forall e_{H_T} \in E_{H_T} : a_{F_{H_T}}(e_{H_T}) \Rightarrow a_{F_{H_T}}(s_{H_T}(e_{H_T})) \wedge a_{F_{H_T}}(t_{H_T}(e_{H_T})) \tag{30}$$

Let e_{H_T} be an edge from E_{H_T}. If an end of e_{H_T} is created in the same derivation step as e_{H_T}, the same feature expression is assigned to both the edge and its end. If the end has been created in a previous step and has been annotated with some feature expression e_F, $a_{F_{H_T}}(e_{H_T}) = e_F \wedge e'_F$ for some feature expression e'_F according to Line 19 of Algorithm 1. In both of these cases, the implication holds for the respective end of the edge, and thus for the conjunction of feature expressions of both ends.

□

Now, we are ready to define *multi-variant derivations*, which relate multi-variant source graphs to multi-variant target graphs:

Definition 20 (*Multi-variant derivation*) Let $H_{S_F} = (H_S, a_{F_{H_S}})$ be a multi-variant source graph, let P be a set of trace-generating STT rules, let $H = H_S \leftarrow H_{TR} \rightarrow H_T$ be an STT graph resulting from a complete STT derivation starting on $H_S \leftarrow \emptyset \rightarrow \emptyset$, and let $a_{F_{H_T}}$ be the annotation function produced by trace-based propagation, applied to H and $a_{F_{H_S}}$. Then, the multi-variant target graph $H_{T_F} = (H_T, a_{F_{H_T}})$ is called *multi-variant derivable* from the multi-variant source graph H_{S_F} ($H_{S_F} \xrightarrow{P_F^*} H_{T_F}$).

3.9 Proof of commutativity

Let us summarize what we already know about single- and multi-variant transformations in our computational model:

- If a multi-variant transformation terminates on a multi-variant source graph, it produces a unique result up to isomorphism. This follows from the functional behavior of single-variant transformations (Proposition 6) and the functional behavior of trace-based propagation (the result is unique if the function SELECT respects the partial order on trace nodes defined by *use* edges).
- If a multi-variant transformation terminates on a multi-variant source graph, a single-variant transformation, when executed on a filtered source graph, terminates, as well (Proposition 7), and also returns a unique result up to isomorphism.

Since filters always terminate and deliver unique results, we know that if the transform-filter path is executed successfully, it will deliver a unique result, which we may compare to the unique result of the terminating execution of the filter-

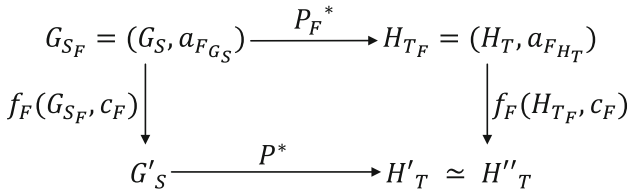

Fig. 32 Commutativity

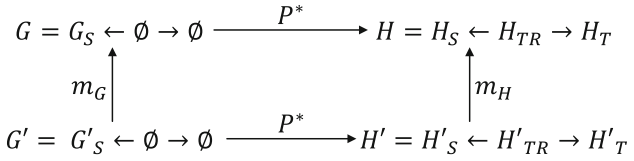

Fig. 33 Application of monomorphism preservation

transform path (assuming that the same filter is applied on both paths).

The following theorem states that the results obtained on different paths are the same (up to isomorphism):

Theorem 1 (Commutativity) *Let F be a feature set, let $G_{S_F} = (G_S, a_{F_{G_S}})$ be a multi-variant source graph, let P be a set of trace-generating STT rules, and let $H_{T_F} = (H_T, a_{F_{H_T}})$ be a multi-variant target graph that is multi-variant derivable from G_{S_F} ($G_{S_F} \xrightarrow{P_F{}^*} H_{T_F}$). Let $f_F : \mathcal{G}_F \times C_F \rightarrow \mathcal{G}$ be a graph filter. Then for every feature configuration $c_F \in C_F$ the following proposition holds:*

Let $G'_S = f_F(G_{S_F}, c_F)$ and $H''_T = f_F(H_{T_F}, c_F)$ denote the filtered source and target graph, respectively. Let H'_T be derivable from G'_S via a single-variant derivation. Then $H'_T \simeq H''_T$ (Fig. 32).

Proof Let $c_F \in C_F$ denote a feature configuration. For the multi-variant derivation, let $G \xrightarrow{P^*} H$ denote the underlying complete STT derivation, where $G = G_S \leftarrow \emptyset \rightarrow \emptyset$ and $H = H_S \leftarrow H_{TR} \rightarrow H_T$. Likewise, for the single-variant derivation let $G' \xrightarrow{P^*} H'$ denote the underlying complete STT derivation, where $G' = G'_S \leftarrow \emptyset \rightarrow \emptyset$ and $H' = H'_S \leftarrow H'_{TR} \rightarrow H'_T$. Since source graphs are not modified in STT derivations, $G_S = H_S$ and $G'_S = H'_S$.

Since $G' \subseteq G$, the identity on G' is a monomorphism $m_G : G' \rightarrow G$. Proposition 5 implies that there is a corresponding monomorphism $m_H : H' \rightarrow H$. Thus, we obtain the diagram of Fig. 33.

Let $m_{H_T} : H'_T \rightarrow H_T$ denote the restriction of m_H to the target graph H'_T, i.e., $m_{H_T} = m_H|_{H'_T}$. We will prove that the range of m_{H_T} coincides exactly with the filtered target graph H''_T:

$$ran(m_{H_T}) = H''_T \tag{31}$$

As a consequence, the monomorphism $m_{H_T} : H'_T \rightarrow H''_T$ is surjective. Thus, it is an isomorphism from H'_T to H''_T, and $H'_T \simeq H''_T$.

Let $m_{EL_{HT}}$ denote the element mapping function of m_{H_T}. To prove (31), we rephrase this statement as follows for the elements from H_T:

$$\forall el_{H_T} \in EL_{H_T} : el_{H_T} \in ran(m_{EL_{H_T}}) \Leftrightarrow v_F(a_{F_{H_T}}(el_{H_T}), c_F) = true \tag{32}$$

Thus, an element belongs to the range of $m_{EL_{H_T}}$ if and only if it passes the filter under the feature configuration c_F.

On the source graph, we already know that an analogous statement holds for the element mapping function m_{EL_G} of the monomorphism m_G because G'_S is obtained from G_S by filtering G_S under the same feature configuration c_F as the target graph H_T:

$$\forall el_{G_S} \in EL_{G_S} : el_{G_S} \in ran(m_{EL_{G_S}}) \Leftrightarrow v_F(a_{F_{G_S}}(el_{G_S}), c_F) = true \tag{33}$$

For $G \xrightarrow{P^*} H$, let $G_0 \ldots G_n$, $p_0 \ldots p_{n-1}$, and $m_0 \ldots m_{n-1}$ be sequences of graphs, rules, and matches satisfying the conditions from Definition 9; likewise for $G' \xrightarrow{P^*} H'$, $G'_0 \ldots G'_{n'}$, $p'_0 \ldots p'_{n'-1}$, and $m'_0 \ldots m'_{n'-1}$, respectively ($n, n' \geq 0$).

Let us consider the sequence $G_0 \ldots G_n$. This sequence is increasing monotonically (Proposition 4), and $G_n = H$. Furthermore, $G_i = G_{S_i} \leftarrow G_{TR_i} \rightarrow G_{T_i}$ (where $G_{S_i} = G_S$ for each $0 \leq i \leq n$). We prove the following statement by induction over i:

$$\forall el_{G_{T_i}} \in EL_{G_{T_i}} : el_{G_{T_i}} \in ran(m_{EL_{H_T}}) \Leftrightarrow v_F(a_{F_{H_T}}(el_{G_{T_i}}), c_F) = true \tag{34}$$

Since $G_{T_n} = H_T$, (34) is equivalent to (32) for $i = n$.

Proof by induction: for $i = 0$, $G_{T_0} = \emptyset$ (no derivation steps have been performed yet). Thus, (34) is trivially satisfied.

For the induction step $i \rightarrow i+1$, let us assume that (34) is valid for index i. Let (p_i, m_i) denote the next derivation step to consider. Furthermore, let SE_i, CE_i, and TE_i denote its source, context, and target elements, respectively. Finally, let $te_i \in TE_i$ be some target element. According to Algorithm 1, te_i is annotated with the following feature expression e_{F_i}:

$$e_{F_i} = \bigwedge \{a_{F_{G_{S_i}}}(se_i) \mid se_i \in SE_i\} \wedge \bigwedge \{a_{F_{G_{T_i}}}(ce_i) \mid ce_i \in CE_i\} \tag{35}$$

Thus, for the feature configuration $c_F \in C_F$ expression e_{F_i} evaluates to true if and only if the feature expressions of

all source and context elements evaluate to true:

$$v_F(e_{F_i}, c_F) = true \Leftrightarrow \\ \forall se_i \in SE_i : v_F(a_{F_{G_{S_i}}}(se_i), c_F) = true \wedge \\ \forall ce_i \in CE_i : v_F(a_{F_{G_{T_i}}}(ce_i), c_F) = true \quad (36)$$

Using (33) and the inductive assumption (34), we may conclude:

$$v_F(e_{F_i}, c_F) = true \Leftrightarrow \\ \forall se_i \in SE_i : se_i \in ran(m_{EL_{G_S}}) \wedge \\ \forall ce_i \in CE_i : ce_i \in ran(m_{EL_{H_T}}) \quad (37)$$

According to Proposition 5, all source elements from SE_i and all context elements from CE_i are members of the ranges of the respective mapping functions if and only if there is a corresponding match (p'_j, m'_j) occurring in $G' \overset{P^*}{\rightarrow} H'$ such that the following conditions hold:

- $p_i = p'_j$.
- m_H maps the elements of the match m'_j to the elements of the match m_i.

Thus, expression e_{F_i} attached to element $te_i \in TE_i$ evaluates to true if and only if the match m_i of the derivation step (p_i, m_i) creating te_i is related to the match of some corresponding derivation step (p'_j, m'_j). According to Proposition 5, the matches correspond to each other if and only if the elements created by the corresponding derivation steps correspond to each other. Altogether, we obtain:

$$v_F(e_{F_i}, c_F) = true \Leftrightarrow te_i \in ran(m_{EL_{H_T}}) \quad (38)$$

This condition holds for each $te_i \in TE_i$. Furthermore, $EL_{G_{T_{i+1}}} = EL_{G_{T_i}} \cup TE_i$. Thus, the following proposition holds:

$$\forall el_{G_{T_{i+1}}} \in EL_{G_{T_{i+1}}} : el_{G_{T_{i+1}}} \in ran(m_{EL_{H_T}}) \Leftrightarrow \\ v_F(a_{F_{H_T}}(el_{G_{T_{i+1}}}), c_F) = true \quad (39)$$

This completes the induction step and the proof as a whole. □

4 Extensions

In the previous sections, we presented a computational model for trace-based propagation of annotations and proved that a multi-variant model transformation conforming to the computational model commutes. The current section presents problems which arise when the computational model is violated, and demonstrates practical solution approaches.

In Sect. 4.1, potential violations of the computational model are enumerated, followed by introducing an illustrative example in Sect. 4.2. Section 4.3 addresses incomplete traces. Section 4.4 investigates the use of incremental transformations for establishing commutativity. Section 4.5 gives a brief summary.

4.1 Computational model violations

First of all, our solution requires to access complete traces but this information is not always made available to the user by transformation engines. As a consequence, two problems may occur: either there is only incomplete information on corresponding elements (*incomplete* or *generation-complete trace*) or there is *no trace* present at all. In addition, the level of detail of the trace may differ from the level of detail of variability annotations. For instance, the trace may record objects only, whereas annotations may decorate the structural features of these objects, too. Below, we focus on the situation when incomplete trace information is still available. Then, our approach can be adapted to this situation.

In the second place, the *properties* of the *transformation rules* are a prerequisite for the correct propagation of annotations. However, transformation rules may violate properties of the computational model, i.e., they may not be functional, monotonic, or local. As it was discussed in Sect. 2.5.4, transformation rules in the tools introduced as representatives are functional, whereas they may not expose a monotonic or local behavior. In particular, locality cannot be guaranteed for any of these tools.

Finally, the computational model assumes batch transformations; commutativity holds for batch transformations only. *Incremental transformations* have not been considered so far.

4.2 Scenario violating commutativity

In order to illustrate violations of commutativity and respective solution approaches, we use a well-known example that deals with the transformation between a *families database* and a *persons database*. This example was proposed originally as part of the ATL [21] transformation zoo.[3] Here, we are concerned only with the backward transformation (persons to families).

On the right-hand side, Fig. 34 displays *metamodels* (in Ecore) for both databases. A persons database contains a flat set of persons with name attributes (being composed of the first name and the family name). The abstract class `Person` is refined into two subclasses for female and male persons, respectively. A families database consists of named families.

[3] http://www.eclipse.org/atl/atlTransformations/#Families2Persons.

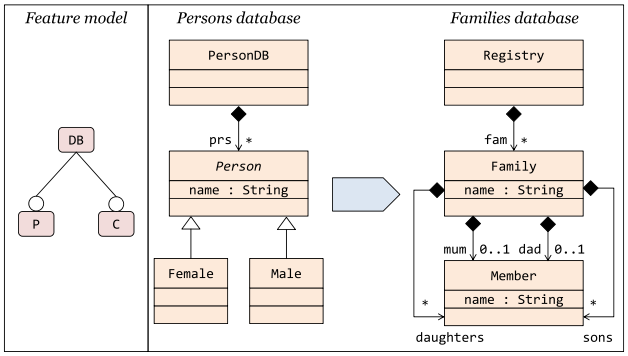

Fig. 34 Metamodels for the persons to families transformation (right) and feature model (left) used in the scenario

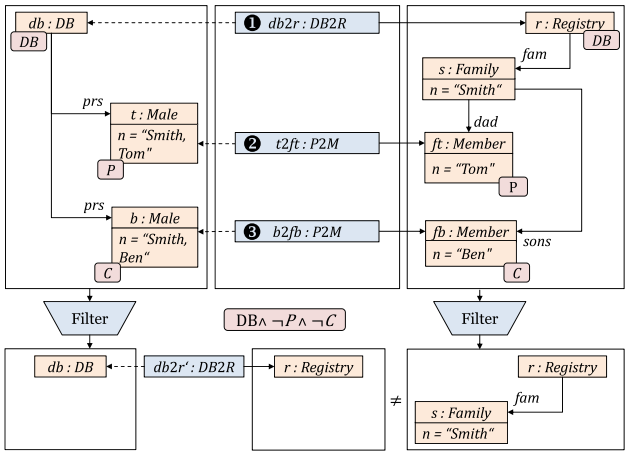

Fig. 35 Transformation of an annotated persons database to an annotated families database using an incomplete trace. Commutativity is violated due to the missing annotation on the `Family` element

Family members carry their first names as attributes. The role of a member is expressed by its incoming containment link. Each family has at most one mum and one dad, and any number of daughters and sons. Note that we assume all multi-valued references to be ordered.

On its left-hand side, Fig. 34 displays a *feature model*, which comprises the mandatory root *DB* and two optional features *P* and *C*, allowing to select a children or parents database only.

The *transformation* of a persons database to a families database is assumed to behave as follows: the first rule application creates the family registry. Thereafter, the persons recorded in the persons database are processed sequentially, according to the order in which they are contained in the `prs` list. For each person, a family member is created. When a `Family` corresponding with the last name of the person is not present yet, a `Family` element is created, and the `Member` is inserted as parent, as `mum` or `dad` depending on its gender. If the family exists but the parent role corresponding with the person's gender is not occupied, the member will be inserted as parent; otherwise, it becomes a child.

4.3 Incomplete traces

For trace-based propagation, the most severe problem consists in missing information. In the following, we assume that the transformation is realized in a tool (e.g., BXtend) that creates 1:1 correspondences between source and target model elements. As a consequence, a trace may be *incomplete* if multiple target elements are created from the same source element, but only one target element participates in the trace. Figure 35 gives an example. As presented on the top of the figure, the trace records only one main corresponding element, i.e., in this use case the `Family` element is not mapped by any trace element. Note that we assume that Tom precedes Ben in the `prs` list. Thus, Tom is transformed first as a dad; next, Ben follows as a son.

When trace-based propagation is applied as described above, all elements missing an annotation are globally visible by default, i.e., in this example the family Smith is part of every filtered target model. Evaluating commutativity gives a negative result in the case of selecting the feature `DB` only, as shown at the bottom of the figure. The family remains in the filtered target model, whereas it is not created by transforming the filtered source model.

For that reason, we propose different strategies to determine *missing annotations* in *partially annotated models* in a post-processing step [17]. These strategies are based on the structure of the target model, which is assumed to have a spanning containment tree. The elements missing an annotation are collected and treated by either of the following strategies:

- *Container* The annotation of the container element is taken, starting to assign the annotations from the top to the bottom.
- *Contained* The annotation of children elements is combined in a *disjunction*, starting to assign the annotations from the bottom to the top. If an element missing an annotation does not contain any elements (leaf of the tree), the annotation $true$ may be assigned.
- *Combined* The annotation of the container and the children is combined in a *conjunction*, i.e., at first the tree is iterated from the top to the bottom and the container expressions are assigned, followed by iterating from the bottom to the top and combining the annotation of the container with the one resulting from the annotations of the children.

Despite a negligibly increased computational effort, we propose to utilize the combined strategy: an element in a

hierarchy is visible if and only if its parent is visible and at least one of its children is visible. This strategy assigns the annotation $DB \wedge (P \vee C)$ to the `Family` element. Applying the 1:1 propagation followed by complementing missing annotations with the post-processing strategy solves the above mentioned problem, i.e., commutativity is achieved in Fig. 35.

Summing it up, the problem of incomplete trace information may be addressed by first exploiting the existing information (e.g., propagating the annotations of 1:1 mappings) and then applying a post-processing strategy. Above, we have argued that the combined strategy should deliver the best results (with respect to commutativity). However, any of the proposed strategies is a heuristic; it cannot guarantee commutativity. Therefore, the post-processing strategy should be selectable by the user. Furthermore, annotations created in the post-processing step should be considered as proposals to be checked and corrected (if necessary) by the user (e.g., by leaving markers on these annotations that may be removed after they have been checked and corrected).

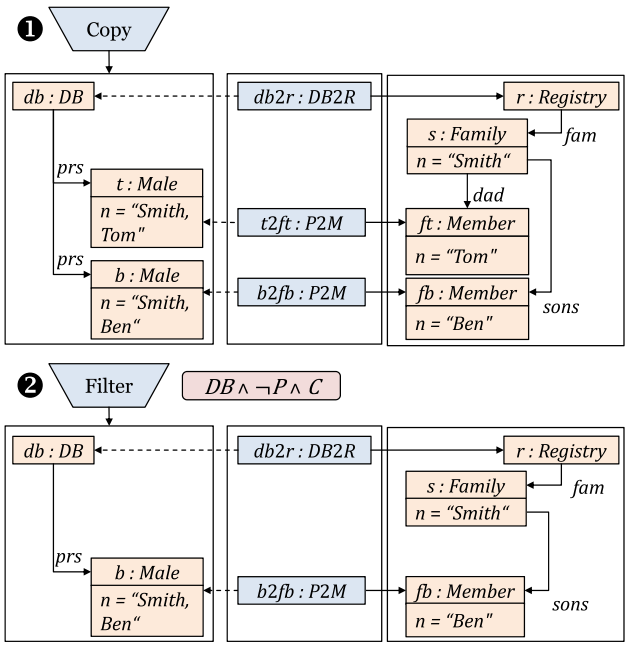

Fig. 36 Sketch of incremental commutativity for handling deletions at the level of application engineering

4.4 Incremental single-variant transformations

The second kind of problem arises when the transformation rules do not conform to the computational model. In particular, in all transformation languages discussed in Sect. 2.5.4 rules are not guaranteed to be local, i.e., their application may depend on conditions that extend beyond the respective matches. As a consequence, a condition which is satisfied on a multi-variant model may not hold on a single-variant model and vice versa. In the case of non-local rules, commutativity may be violated.

Let us assume that family Smith carries, based on the combined strategy, the expression $DB \wedge (P \vee C)$. For evaluating commutativity all feature configurations are created. In the case P and C are deselected, only the registry remains in the filtered target model. The same holds for the database in the persons database, which in turn is transformed into the family registry. Thus, commutativity is achieved for the feature configuration $DB \wedge \neg P \wedge \neg C$. The same holds true when parents are selected, regardless of whether children are selected or not (feature configurations $DB \wedge P \wedge \neg C$ and $DB \wedge P \wedge C$). In contrast, when a children database without parents is selected ($DB \wedge \neg P \wedge C$), in the target model derived from the multi-variant target model Ben remains as son of the family Smith and as male person in the filtered source model. However, when transforming the filtered source model, a family Smith is created and Ben is inserted as dad. Accordingly, commutativity is violated since Ben is mapped to a son on the transform-filter path and to a dad on the filter-transform path. Altogether, the combined strategy improves commutativity (the number of commuting cases is increased), but does not resolve all violations.

It is important to notice that this problem is fundamental: for the commutativity criterion which has been proposed so far, the problem cannot be solved by any post-processing approach that annotates the result of a single-variant transformation. On the transform-filter path, Ben is inserted as a son (if Ben passes the filter on the target model). On the filter-transform path, Ben might be inserted as a parent. Filtering a target model in which Ben occurs as a son cannot result in a model in which Ben occurs as a parent. No matter how we assign annotations to target model elements: commutativity is violated in the case described above. Note that this problem is caused by *non-local rules*: the rule for transforming a person to a member has to check the context in the target model. It depends on this context how the person is transformed.

In the following, we propose a solution that may further improve commutativity under the prerequisite that single-variant transformations and filters may be executed *incrementally*. Rather than relying on batch transformations only, commutativity is checked with the help of incremental transformations. In this proposal, commutativity is evaluated as follows (Fig. 36, illustrating steps 1 and 2 below):

1. In an initializing step, the multi-variant source model is copied to the level of application engineering (stripping the variability annotations). Subsequently, this 150% source model is transformed (in batch mode) into a 150% target model.
2. The filter-transform path is executed as follows: first, the selected filter is applied incrementally to the 150% source

model. This means that all elements not selected by the filter are deleted. Subsequently, an incremental transformation is executed against the 150% target model. As a consequence, deletions on the source model are propagated to the target model.
3. On the transform-filter path, the multi-variant transformation is applied to the multi-variant source model in batch mode. Subsequently, the resulting multi-variant target model is filtered in batch mode.
4. The single-variant target models obtained on both paths are compared to each other.

In our example, this approach solves the violating case for the feature configuration $DB \wedge \neg P \wedge C$. Since the parent is deselected, Tom is deleted from the source model, and this deletion is propagated to the target model. Ben is not affected by this change: the incremental transformation still knows that the male Ben Smith corresponds to the son Ben of the family Smith. Thus, it does not create a new parent (as it would happen in a batch transformation) but keeps Ben as son. Altogether, the proposed use of incremental transformations results in commuting transformations for all valid feature configurations.

4.5 Summary

In this section, we have sketched extensions to trace-based propagation that may be employed if certain properties of the computational model are not satisfied. The first extension adds missing variability annotations in the case of incomplete traces. The second extension exploits incremental transformations to increase stability of transformations. Both extensions may improve the behavior of multi-variant transformations, as demonstrated by the given example. However, these extensions are heuristics that work for certain transformations, but may still fail in other cases. Altogether, commutativity is improved, but not guaranteed.

5 Related work

We have already discussed related work performed by other researchers at various locations in this paper (Sects. 1.4, 2.2.3, 2.3.2, 2.5.4). This section brings these discussions together and takes further aspects into account that have not been considered so far.

Section 5.1 addresses multi-variant transformations. Section 5.2 is concerned with the computational model developed in Sect. 3. Section 5.3 focuses on the proof of commutativity.

5.1 Multi-variant transformations

The term "multi-variant model transformation" is ambiguous and may be parsed in two ways: transformation of multi-variant models, the focus of our work, and multi-variant transformation of models, as addressed in [42,43]. Likewise, transformations of product lines [44], covering both feature and domain models, goes beyond the scope of our work.

Approaches to transforming multi-variant models may be classified according to three criteria: scope, abstraction, and realization (Fig. 11). In Sect. 2.2.3, we have classified approaches from the literature according to these criteria (Table 1) and have evaluated them (Table 2) against the requirements stated in Sect. 2.2.1. Among these approaches, trace-based propagation is unique inasmuch as it may be applied independently of the language in which single-variant transformations are written—provided that transformations record persistent traces (see also Sect. 1.4). Thus, trace-based propagation may be applied in a heterogeneous MDPLE environment such as Feature Mapper [19] or Famile [8,9] and meets the requirements for genericity and reuse.

Traces may be classified according to three criteria: type, completeness, and granularity (Fig. 13). The traces created by a set of different tools used as running examples throughout this paper (ATL/EMFTVM [45], BXtend [6], eMoflon [28], and medini QVT [20]) are classified with respect to this taxonomy in Table 3; see also Sect. 2.3.2. For achieving commutativity, complete and fine-grained traces are required (Sect. 2.5.1).

5.2 Computational model

A core contribution of this paper is a computational model for single-variant transformations that guarantees commutativity of multi-variant transformations realized with the help of trace-based propagation. The properties of the computational model have been summarized in Sect. 2.5.1: a transformation is performed out-place in batch mode by applying all transformation rules to all matches. Each rule is functional, monotonic, and local. Finally, traces must be complete and fine-grained.

The computational model provides a precise characterization of the conditions under which commutativity is satisfied. Commutativity holds as long as transformations conform to the computational model. However, contemporary transformation languages and tools usually allow to write non-conforming transformations. For the tools used as running examples, Table 4 in Sect. 2.5.4 states which properties are satisfied.

In our previous work [47], we presented a computational model that considers models as sets of elements (rather than graphs). The set-based model and the graph-based model share the same properties regarding transformations, rules,

and traces. The set-based and the graph-based formalizations are considerably different, but result in similar propositions concerning computational behavior and commutativity. While it is possible to consider models as flat sets of elements, the graph-based formalization appears to be more natural and structured. Furthermore, it facilitates the comparison to other graph-based computational models.

With respect to the definition of graphs in the computational base model (Sects. 3.2–3.4), our definition is slightly more general than the definition of graphs in algebraic graph transformation [12] inasmuch as it allows for higher-order edges. On the other hand, our rules are monotonic and local. In contrast, algebraic graph transformation allows for deletion of nodes and edges as well as application conditions. As a consequence, our rules are parallel independent on all matches. Thus, transformations are confluent and exhibit functional behavior. Under certain conditions, these properties hold for algebraic graph transformations, as well (Sect. 3.4 in [12]). Furthermore, the embedding theorem that is stated in [12] resembles Proposition 5, which states that each rule applied in a derivation starting from some graph G may be mapped to a corresponding rule application in a derivation starting from a graph G' including G.

With respect to single-variant transformations defined on top of the computational base model, we may relate our work to triple graph grammars (TGGs [38]). In TGGs, out-place graph transformations are specified with the help of monotonic grammar rules which extend the source graph, the correspondence graph, and the target graph simultaneously. From these synchronous rules, directed rules for forward and backward transformations are derived automatically. Our source-to-target rules correspond to directed TGG rules. However, the computational model of TGGs differs in various aspects: while rules are both monotonic and local in the original TGG proposal, contemporary tools such as eMoflon support negative application conditions, which violate locality. Furthermore, rather than applying each rule to each match, the source graph is covered by rule applications. As a consequence, rules compete for matches, and the TGG engine has to resolve conflicts between rule applications. Altogether, a TGG-based transformation may be non-functional, which is not possible in our computational model.

5.3 Commutativity

The proof of commutativity for the graph-based computational model is conceptually similar to the proof presented in our previous work [47]. To the best of our knowledge, the first formalization of the commutativity criterion as well as a formal proof of commutativity was given in [34], which lifts in-place graph transformation rules to multi-variant graphs (while our approach considers out-place batch transformations). The lifting approach is more general than our approach inasmuch as deletion of nodes and edges as well as negative application conditions are allowed in graph transformation rules. When a rule is applied, the annotations attached to graph elements are calculated according to a formula which takes deletions and negative application conditions into account. In trace-based propagation, neither deletions nor negative application conditions can be taken into account because the actual rules driving the transformation are not known.

Our proof of commutativity differs from the respective proof for the lifting approach in the following respect: in [34], commutativity is proved for a single rule application: application of a lifted, multi-variant rule followed by a filter yields the same result as applying the filter first, followed by an application of the single-variant rule. In contrast, we have to consider complete derivations. This applies both to the central Proposition 5, which relates derivations on graphs G and G' including G, and to the final Theorem 1, which proves commutativity.

6 Conclusion

We conclude this paper with a summary (Sect. 6.1) and an outlook on future work (Sect. 6.2).

6.1 Summary

Trace-based propagation of variability annotations has been developed to serve a practical need: to extend single- to multi-variant transformations in a heterogeneous MDPLE environment such as Famile or Feature Mapper, where different languages and tools may be employed for defining and executing model transformations. Trace-based propagation is a generic gray-box approach: it needs only access to the trace of a single-variant transformation via a generic interface that may be implemented for different languages and tools. In this way, trace-based propagation provides a low-cost solution for implementing multi-variant transformations in a heterogeneous MDPLE environment.

The main contribution of this paper consists in the promotion of trace-based propagation from a heuristic to an approach with a proven correctness property. The notion of correctness is defined with the help of commutativity: a multi-variant transformation is correct if filters and transformations commute. Executing a multi-variant transformation followed by a filter executed on the target model must yield the same result as filtering the source model first, followed by a single-variant transformation. Commutativity must hold for each filter—provided that the same filter is applied to the source and the target model.

To prove commutativity, we have developed a graph-based computational model for single-variant transformations that satisfies various properties with respect to rules, transformations, and traces. The computational model assumes that rules are functional, local, and monotonic, that all rules are applied to all matches, and that fine-grained and complete traces are recorded when rules are executed. Based on this computational model, we have formalized trace-based propagation and proved that commutativity is satisfied. In this way, we have identified clearly under which conditions trace-based propagation guarantees commutativity.

Trace-based propagation may still be used when the prerequisites underlying the computational model are not satisfied. As we have discussed in this paper, transformation languages may not conform to the computational model, and traces may be incomplete or coarse-grained. Even then, a large fraction of the annotations generated by trace-based propagation may be correct, minimizing the need for manual adjustments. Furthermore, we proposed various extensions that further improve the results of trace-based propagation. Altogether, we consider trace-based propagation a pragmatic approach that may be used to provide for multi-variant transformations, serving the needs of both users and developers of heterogeneous MDPLE environments.

6.2 Future work

Future work on trace-based propagation should address the following issues:

6.2.1 Transformation languages and tools

The proof of commutativity relies on a formal graph-based computational model (Sect. 3). This model may serve as the foundation for developing a new transformation language and tool that is guaranteed to establish commutativity of filters and transformations.

As we discussed in Sect. 2.5.4, existing languages and tools break the computational model introduced in Sect. 2.5.1. The design of (single-variant) transformation languages is driven by expressiveness as one of the most important goals: which kinds of transformations may be specified in the respective language? For this reason, existing transformation languages go beyond our computational model by supporting incremental in addition to batch transformations, rules that allow updates and deletions in the target model or permit checking of conditions, and explicit control structures for rule applications.

Rather than making existing languages conform to the computational model (which would restrict their expressiveness), it should be investigated whether subsets of these languages may be identified that exhibit conformance. Thus, filters and transformations would commute as long as only those language constructs are used that belong to the respective subset.

As far as traces are concerned, language and tool support should still be improved. The computational model requires traces to be both complete and fine-grained. None of the investigated languages and tools fully meets these requirements. Therefore, tools should be extended such that they produce complete and fine-grained traces. Since traces may be employed also for other purposes (e.g., incremental transformations), the effort for performing this work should pay off.

6.2.2 Extensions of trace-based propagation

In Sect. 4, we presented still ongoing work on extensions of trace-based propagation. This work addresses two problems:

- Incomplete traces result in variability annotations of target model elements that are not correct with respect to the commutativity criterion. We proposed several strategies for correcting variability annotations in a post-processing step. These strategies need to be explored further.
- So far, we have performed only initial work on incremental transformations (see Sect. 4.4). In an incremental setting, all involved transformations would operate incrementally: multi-variant transformations in domain engineering, filters for the derivation of products, and single-variant transformations in application engineering. Accordingly, the commutativity criterion, the algorithm for trace-based propagation, and the underlying computational model would have to be generalized to the incremental case.

6.2.3 Evaluation

Finally, our work on trace-based propagation needs to be evaluated on a larger scale. So far, we have considered fairly small synthetic sample transformations, such as the transformation from Gantt diagrams to CPM networks that served as a running example, and the transformation from persons databases to families databases. In addition, we will apply trace-based propagation to larger, industrially relevant transformations, such as model to code transformations, transformations between UML models and Ecore models, or object-relational mappings.

For evaluating multi-variant transformations, we have proposed a set of metrics and a generic framework that is described in [16] and is currently being implemented on top of the BenchmarX framework [1].

Acknowledgements Open Access funding provided by Projekt DEAL. The authors greatly appreciate the constructive comments of the unknown reviewers.

References

1. Anjorin, A., Buchmann, T., Westfechtel, B., Diskin, Z., Ko, H.-S., Eramo, R., Hinkel, G., Samimi-Dehkordi, L., Zündorf, A.: Benchmarking bidirectional transformations: theory, implementation, application, and assessment. Softw. Syst. Model. (2019) (Online first)
2. Apel, S., Kästner, C.: Virtual separation of concerns–a second chance for preprocessors. J. Object Technol. **8**(6), 59–78 (2009)
3. Barroca, B., Lúcio, L., Amaral, V., Félix, R., Sousa, V.: DSLTrans: a Turing incomplete transformation language. In: Malloy, B., Staab, S., van den Brand, M. (eds.) Proceedings of the Third International Conference on Software Language Engineering (SLE 2010). Lecture Notes in Computer Science, Eindhoven, The Netherlands, Oct 2010, vol. 6563, pp. 296–305. Springer
4. Batory, D.S., Sarvela, J.N., Rauschmayer, A.: Scaling step-wise refinement. IEEE Trans. Softw. Eng. **30**(6), 355–371 (2004)
5. Beuche, D.: Pure: variants. In: Capilla, R., Bosch, J., Kang, K.C. (eds.) Systems and Software Variability Management. Concepts, Tools and Experiences, pp. 173–182. Springer, Berlin (2013)
6. Buchmann, T.: BXtend—a framework for (bidirectional) incremental model transformations. In: Hamoudi, S., Pires, L.F., Selic, B. (eds.) Proceedings of the 6th International Conference on Model-Driven Engineering and Software Development (MODELSWARD 2018), pp. 336–345, Funchal, Madeira, Portugal, Feb 2018. SCITEPRESS
7. Buchmann, T., Greiner, S.: Managing variability in models and derived artefacts in model-driven software product lines. In: Hamoudi, S., Pires, L.F., Selic, B. (eds.) Proceedings of the 6th International Conference on Model-Driven Engineering and Software Development (MODELSWARD 2018), pp. 326–335, Funchal, Madeira, Portugal, Feb 2018. SCITEPRESS
8. Buchmann, T., Schwägerl, F.: Ensuring well-formedness of configured domain models in model-driven product lines based on negative variability. In: Proceedings of the 4th International Workshop on Feature-Oriented Software Development, FOSD 2012, pp. 37–44, Dresden, Germany, Sep 2012. ACM
9. Buchmann, T., Schwägerl, F.: FAMILE: tool support for evolving model-driven product lines. In: Störrle, H., Botterweck, G., Bourdellès, M., Kolovos, D., Paige, R., Roubtsova, E., Rubin, J., Tolvanen, J.-P. (eds.) Joint Proceedings of co-located Events at the 8th European Conference on Modelling Foundations and Applications, CEUR WS, pp. 59–62, Kongens Lyngby, Denmark, July 2012. Technical University of Denmark (DTU)
10. Cicchetti, A., Di Ruscio, D., Eramo, R., Pierantonio, A.: JTL: a bidirectional and change propagating transformation language. In: Malloy, B., Staab, S., van den Brand, M. (eds.) Proceedings of the Third International Conference on Software Language Engineering (SLE 2010). Lecture Notes in Computer Science, vol. 6563, pp. 183–202, Eindhoven, The Netherlands, Oct 2010. Springer
11. Czarnecki, K., Helsen, S.: Feature-based survey of model transformation approaches. IBM Syst. J. **45**(3), 621–645 (2006)
12. Ehrig, H., Ehrig, K., Prange, U., Taentzer, G.: Fundamentals of Algebraic Graph Transformation. Monographs in Theoretical Computer Science. An EATCS Series. Springer, Berlin (2006)
13. Famelis, M., Lúcio, L., Selim, G., Di Sandro, A., Salay, R., Chechik, M., Cordy, J.R., Dingel, J., Vangheluwe, H., Ramesh S.: Migrating automotive product lines: a case study. In: Kolovos, D., Wimmer, M. (eds.) Proceedings of the 8th International Conference on Theory and Practice of Model Transformations (ICMT 2015). Lecture Notes in Computer Science, vol. 9152, pp. 82–97, L'Aquila, Italy, July 2015. Springer
14. Greiner, S., Schwägerl, F., Westfechtel, B.: Realizing multi-variant model transformations on top of reused ATL specifications. In: Pires, L.F., Hammoudi, S., Selic, B. (eds.) Proceedings of the 5th International Conference on Model-Driven Engineering and Software Development (MODELSWARD 2017), pp. 362–373, Porto, Portugal, Feb 2017. SCITEPRESS
15. Greiner, S., Westfechtel, B.: Generating multi-variant Java source code using generic aspects. In: Hamoudi, S., Pires, L.F., Selic, B. (eds.) Proceedings of the 6th International Conference on Model-Driven Engineering and Software Development (MODELSWARD 2018), pp. 36–47, Funchal, Madeira, Portugal, Feb 2018. SCITEPRESS
16. Greiner, S., Westfechtel, B.: Generic framework for evaluating commutativity of multi-variant model transformations. In: Hammoudi, S., Pires, L.F., Selic, B. (eds.) Proceedings of the 7th International Conference on Model-Driven Engineering and Software Development (MODELSWARD 2019), vol. 1, pp. 157–168, Prague, Czech Republic, Feb 2019. SCITEPRESS
17. Greiner, S., Westfechtel, B.: On determining variability annotations in partially annotated models. In: Proceedings of the 13th International Workshop on Variability Modelling of Software-Intensive Systems (VAMOS 2019), pp. 17:1–17:10, Leuven, Belgium, Feb 2019. ACM
18. Halmos, P.R.: Naive Set Theory. Undergraduate Texts in Mathematics. Springer, New York (1974)
19. Heidenreich, F.: Towards systematic ensuring well-formedness of software product lines. In: Proceedings of the 1st International Workshop on Feature-Oriented Software Development (FOSD 2009), pp. 69–74, Denver, CO, USA, Oct 2009. ACM
20. ikv++ technologies. medini QVT. ikv++ technologies, 2019. http://projects.ikv.de/qvt. Accessed 19 De 2019
21. Jouault, F., Allilaire, F., Bézivin, J., Kurtev, I.: ATL: a model transformation tool. Sci Comput Program **72**, 31–39 (2008). Special Issue on Experimental Software and Toolkits (EST)
22. Kang, K.C., Cohen, S.G., Hess, J.A., Novak, W.E., Peterson, A.S.: Feature-oriented domain analysis (FODA) feasibility study. Technical report CMU/SEI-90-TR-21, Carnegie-Mellon University, Software Engineering Institute (1990)
23. Kästner, C.: Virtual separation of concerns: towards preprocessors 2.0. Ph.D. thesis, University of Magdeburg (2010)
24. Kerzner, H.: Project Management: A Systems Approach to Planning, Scheduling, and Controlling, 12th edn. Wiley, Hoboken (2017)
25. Krueger, C.W., Clements, P.: Feature-based systems and software product line engineering with gears from BigLever. In: Collet, P., Guo, J., Martinez, J., Seidl, C., Rubin, J., Diaz, O., Mukelabai, M., Berger, T. (eds.) Proceeedings of the 22nd International Systems and Software Product Line Conference (SPLC 2018), vol. 2, pp. 1–4, Gothenburg, Sweden, Sept 2018. ACM
26. Kühne, T.: Matters of (meta-)modeling. Softw. Syst. Model. **5**(4), 369–385 (2006)
27. Le Calvar, T., Jouault, F., Chhel, F., Clavreul, M.: Efficient ATL incremental transformations. J. Object Technol. **18**(3), 2:1–17 (2019)

28. Leblebici, E., Anjorin, A., Schürr, A.: Developing eMoflon with eMoflon. In: Di Ruscio, D., Varró, D. (eds.) Proceedings of the 7th International Conference on Theory and Practice of Model Transformations (ICMT 2014). Lecture Notes in Computer Science, vol. 8568, pp. 138–145, York, UK, July 2014. Springer
29. Lopez-Herrejon, R.E., Batory, D.S.: A standard problem for evaluating product-line methodologies. In: 3rd International Conference on Generative and Component-Based Software Engineering (GCSE 2001). Lecture Notes in Computer Science, vol. 2186, pp. 10–24, Erfurt, Germany, Sep 2001. Springer
30. Mezini, M., Beuche, D., Moreira, A. (eds.): Proceedings 1st international Workshop on Model-Driven Product Line Engineering (MDPLE 2009), Twente, The Netherlands, June 2009. CTIT Proceedings WP09-10
31. Object Management Group: Meta Object Facility (MOF) Version 2.5. Object Management Group, Needham, MA, formal/2015-06-05 edition (2015)
32. Object Management Group: Meta Object Facility (MOF) 2.0 Query/View/Transformation Specification Version 1.3. Object Management Group, Needham, MA, formal/2016-06-03 edition (2016)
33. Pohl, K., Böckle, G., van der Linden, F.: Software Product Line Engineering: Foundations, Principles and Techniques. Springer, Berlin (2005)
34. Salay, R., Famelis, M., Rubin, J., Di Sandro, A., Chechik, M.: Lifting model transformations to product lines. In: Proceedings of the 36th International Conference on Software Engineering (ICSE 2014), pp. 117–128, Hyderabad, India, May 2014. ACM Press
35. Samimi-Dehkordi, L., Zamani, B., Kolahdouz-Rahimi, S.: EVL+Strace: a novel bidirectional transformation approach. Inf. Softw. Technol. **100**, 47–72 (2018)
36. Schaefer, I., Bettini, L., Bono, V., Damiani, F., Tanzarella, N.: Delta-oriented programming of software product lines. In: Bosch, J., Lee, J. (eds.) Software Product Lines: Going Beyond, 14th International Conference (SPLC 2010). Lecture Notes in Computer Science, vol. 6287, pp. 77–91, Jeju Island, South Korea, Sep 2010. Springer
37. Schmidt, D.C.: Guest editor's introduction: model-driven engineering. Computer **39**(2), 25–31 (2006)
38. Schürr, A.: Specification of graph translators with triple graph grammars. In: Proceedings of the 20th International Workshop on Graph-Theoretic Concepts in Computer Science (WG 1994). Lecture Notes in Computer Science, vol. 903, pp. 151–163, Herrsching, Germany, 1995. Springer
39. Schwägerl, F., Buchmann, T., Westfechtel, B.: Multi-variant model transformations—a problem statement. In: Maciaszek, L., Filipe, J. (eds.) Proceedings of the 11th International Conference on the Evaluation of Novel Approaches to Software Engineering (ENASE 2016), pp. 203–209, Rome, Italy, April 2016. SCITEPRESS
40. Sijtema, M.: Introducing variability rules in ATL for managing variability in MDE-based product lines. In: Del Fabro, M.D., Jouault, F., Kurtev, I. (eds.) Proceedings of the 2nd International Workshop on Model Transformation with ATL (MtATL 2010), CEUR Workshop Proceedings, pp. 39–49, Malaga, Spain, June 2010. RWTH Aachen University
41. Steinberg, D., Budinsky, F., Paternostro, M., Merks, E.: EMF Eclipse Modeling Framework. The Eclipse Series, 2nd edn. Addison-Wesley, Boston (2009)
42. Strüber, D., Peldzsus, S., Jürjens, J.: Taming multi-variability of software product line transformations. In: Russo, A., Schürr, A. (eds.) Proceedings of the 21st International Conference on Fundamental Approaches to Software Engineering (FASE 2018). Lecture Notes in Computer Science, vol. 10802, pp. 337–355, Thessaloniki, Greece, Apr 2018. Springer
43. Strüber, D., Schulz, S.: A tool environment for managing families of model transformation rules. In: Echahed, R., Minas, M. (eds.) Proceedings of the 9th International Conference on Graph Transformation (ICGT 2016). Lecture Notes in Computer Science, vol. 9761, pp. 89–101, Vienna, Austria, July 2016. Springer
44. Taentzer, G., Salay, R., Strüber, D., Chechik, M.: Transformations of software product lines: a generalizing framework based on category theory. In: Proceedings of the ACM/IEEE 19th International Conference on Model Driven Engineering Languages and Systems (MODELS 2017), pp. 101–111, Saint-Malo, France, Oct 2017. ACM Press
45. Wagelaar, D., Iovino, L., Di Ruscio, D., Pierantonio, A.: Translational semantics of a co-evolution specific language with the EMF transformation virtual machine. In: Hu, Z., de Lara, J. (eds.) Proceedings of the 5th International Conference on Theory and Practice of Model Transformations (ICMT 2012). Lecture Notes in Computer Science, vol. 7307, pp. 192–207, Prague, Czech Republic, May 2012. Springer
46. Westfechtel, B.: Case-based exploration of bidirectional transformations in QVT relations. Softw. Syst. Model. **17**(3), 989–1029 (2018)
47. Westfechtel, B., Greiner, S.: From single- to multi-variant model transformations: trace-based propagation of variability annotations. In: Proceedings of the 21th ACM/IEEE International Conference on Model Driven Engineering Languages and Systems (MODELS 2018), pp. 46–56, Copenhagen, Denmark, Oct 2018. ACM

Bernhard Westfechtel received his diploma degree from University of Erlangen-Nuremberg in 1983 and his doctoral as well as his habilitation degree (all in computer science) from RWTH Aachen University in 1991 and 1999, respectively. Since 2004, he has been a full professor of computer science at University of Bayreuth. His current research interests include graph transformations, model-driven engineering, software product line engineering, and software configuration management.

Sandra Greiner In 2015 Sandra Greiner earned her master's degree in computer science from University of Bayreuth and commenced her Ph.D. studies in the same year at the Chair of Software Engineering, University of Bayreuth. Her research interests are dedicated to the development of software-intensive systems, in particular software product lines, in a model-driven way. A special focus lies on the investigation of model transformations and their technology.