Termination and Expressiveness of Execution Strategies for Networks of Bidirectional Model Transformations

HEIKO KLARE and JOSHUA GLEITZE, KASTEL – Institute of Information Security and Dependability, Karlsruhe Institute of Technology, Germany

When developers describe a software system with multiple models, such as architecture diagrams, deployment descriptions, and source code, these models must represent the system in a uniform way, i.e., they must be and stay *consistent*. One means to automatically preserve consistency after changes to models are model transformations, of which bidirectional transformations that preserve consistency between two models have been well researched. To preserve consistency between multiple models, such transformations can be combined to networks. When transformations are developed independently and reused modularly, the resulting network can be of arbitrary topology. For such networks, no universal strategy exists to orchestrate the execution of transformations such that the resulting models are consistent.

In this article, we prove that termination of such a strategy can only be guaranteed if it is incomplete, i.e., if it is allowed to fail to restore consistency for some changes although an execution order of transformations exists that yields consistent models. We propose such a strategy, for which we prove termination and show that and why it makes it easier for users of model transformation networks to understand the reasons whenever the strategy fails. In addition, we provide a simulator for the comparison of different execution strategies. These findings help transformation developers and users in understanding when and why they can expect the execution of a transformation network to terminate and when they can even expect it to succeed. Furthermore, the proposed strategy guarantees them termination and supports them in finding the reason whenever it is not successful.

CCS Concepts: • **Software and its engineering** → **Consistency**; *Model-driven software engineering*; *Abstraction, modeling and modularity*; Agile software development; Software evolution;

Additional Key Words and Phrases: Model transformation, model consistency, bidirectional transformation, transformation network, transformation execution strategy, transformation termination

ACM Reference format:
Heiko Klare and Joshua Gleitze. 2023. Termination and Expressiveness of Execution Strategies for Networks of Bidirectional Model Transformations. *Form. Asp. Comput.* 35, 3, Article 15 (September 2023), 35 pages.
https://doi.org/10.1145/3543845

This work was supported by funding from the topic *Engineering Secure Systems* of the Helmholtz Association (HGF) and by KASTEL Security Research Labs (46.23.03).
Authors' address: H. Klare and J. Gleitze, KASTEL – Institute of Information Security and Dependability, Karlsruhe Institute of Technology, Am Fasanengarten 5, 76228, Karlsruhe, Germany; emails: klare@kit.edu, joshua.gleitze@alumni.kit.edu.

0934-5043/2023/09-ART15
https://doi.org/10.1145/3543845

1 INTRODUCTION

The development of systems, like software systems, involves the creation and maintenance of multiple models. Each of these models is tailored to the needs of a specific role according to its concerns. Such a concern may be a specific functional part of the system relevant for a person, or a specific property, such as the architecture, the implementation, or the deployment, each presented in its own type of model such as a UML diagram or program code. Consider the development of a mobility system that provides a unified way for customers to make a booking for a journey using different types of transportation provided by different mobility provider companies, such as a train, bus, taxi, and bike-sharing company. Then each of the companies has their own software components or subsystem to integrate into this integrated system, such that each developer may only be interested in information about one subsystem. In addition, different roles may use different tools and according models, such as an architect using UML for a component-based and object-oriented design model or a developer using Java code. Developers may also use API specifications of their web services, which are even shared across the subsystems to be used in a uniform way.

All those models taken together should describe a coherent system and not contain contradictory information. We say that the models should be *consistent*. This comprises both the consistency of models for the different subsystems, e.g., by using the same API specifications for their interoperability, but also the consistency of different models for the same subsystems, such as the architecture being consistent with the implementation. Automatic detection and resolution of inconsistencies is, however, still poorly addressed in current development processes that involve a variety of models, such as automotive development [Guissouma et al. 2018].

Automatically maintaining consistency may be achieved by periodic validations of constraints and the resolution of detected violations or by continuously preserving it. A popular and industrially applied means [Weidmann and Sauer 2020] to preserve consistency are *incremental model transformations*, which update models based on information that was changed in one of them, such that performing changes to consistent models leads to consistent models again by executing these transformations. While there has been significant research on model transformations themselves, particularly on binary transformations keeping two models consistent, maintaining consistency of multiple models is less researched [Cleve et al. 2019]. There are approaches for multiary model transformations, which can transform between multiple models by means of a single transformation. Nevertheless, one will likely also want to be able to combine multiple transformations—binary or multiary—to maintain consistency, creating a *transformation network*. Unlike using a single, overarching transformation, defining a network makes it possible to reuse modular ones [Klare 2021], and the possibility for reuse is essential for the productive use of transformations, in particular in industrial contexts [Bruel et al. 2020]. Additionally, knowledge about consistency between certain types of models is often distributed across domain experts [Klare 2018]. This can be accommodated by transformation networks, because every domain expert can define transformations independently and according to their view on consistency.

When transformations are not developed for a specific network but are meant to be reused across multiple networks, they cannot be aligned with each other, and the resulting network may be of arbitrary topology. To the best of the authors' knowledge, no strategy that determines an execution order of transformations to maintain consistency in a network with arbitrary topology has been presented yet. Existing work proposes, for example, defining an execution order explicitly [von Pilgrim et al. 2008; Vanhooff et al. 2007] or deriving a topological order [Stevens 2020b]. Most approaches restrict the supported kinds of network topologies to such in which each transformation only needs to be executed once.

In this article, we research properties and limitations of a universal strategy that executes a transformation network of arbitrary topology. We show that strategies that apply each transformation only once or a fixed number of times are not useful in practice. At the other end of the spectrum, we prove that not limiting the number of transformation executions does, in general, lead to non-termination. Based on the insight that a universal execution strategy must be incomplete if termination shall be ensured, i.e., that we have to accept that it may not find an execution order yielding consistent models whenever it exists, we derive a useful strategy. We show that this strategy helps developers to find the cause whenever the strategy fails to yield consistent models. In addition, we provide a simulator that eases the evaluation and comparison of different execution strategies, including the proposed one, to, e.g., find one that fits better for specific networks or even for specific requirements or properties to ensure. In detail, we make the following five contributions:

Formalisation *(C1)*: We formalise transformation networks and execution strategies to precisely define their expected properties.

Incompleteness Proof *(C2)*: We prove that a universal execution strategy must be incomplete to avoid non-termination.

Strategy Design *(C3)*: We propose an execution strategy that improves traceability whenever no consistent models are found.

Traceability Improvement *(C4)*: We discuss why and under which conditions the proposed strategy improves traceability whenever no consistent models are found.

Strategy Simulator *(C5)*: We provide a simulator that can be used to examine and compare different execution strategies on different example networks.

The contributions establish fundamental knowledge about the design space of network execution strategies, their undecidability, and difficulties in reducing incompleteness. The proposed strategy helps transformation network developers and users to find the reasons when an execution does not yield consistent models, and the provided simulator eases the investigation of properties of different execution strategies.

This article is an extended version of a paper published in the 2021 proceedings of the ***Fundamental Approaches to Software Engineering*** **(FASE)** conference [Gleitze et al. 2021]. Contributions *C4* (Section 6) and *C5* (Section 7) have been added, and examples, further discussion, and evidence have been extended or added throughout. The contributions of the original paper have recently also been published in the dissertation of Klare [2021] together with some of the extensions we have made in this article.

In this article, we first introduce the problem statement in more detail (Section 2) and embed our contributions into related work (Section 3). We discuss the design space for execution strategies and the limitations of its extreme cases (Section 4) before proposing a practically usable execution strategy based on the insights of the design space discussion (Section 5). The strategy is designed to improve traceability whenever it fails, and we show how it fulfils this property in Section 6. After giving an overview of a publicly available simulator for evaluating different execution strategies (Section 7), we conclude our work in Section 8.

2 PROBLEM STATEMENT

In this section, we will further motivate our research by giving an example and clarifying its context. We provide a simple but expressive enough formalisation for transformation networks and execution strategies to generate a common understanding and formal basis for transformation network orchestration, constituting our contribution *C1*. Finally, we use the formalisation to precisely define our problem statement.

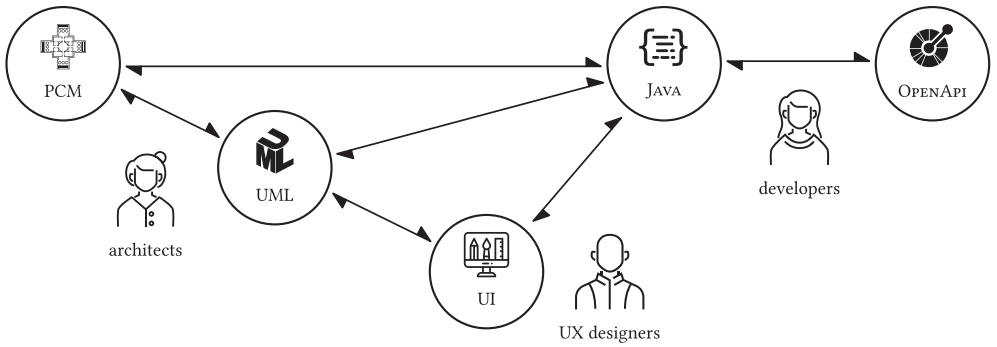

Fig. 1. Example for a transformation network between different types of models in software development.

2.1 Motivating Example

Figure 1 describes a software project whose contributors take the roles of architects, developers, and **user experience (UX)** designers. One person can take multiple roles, but every role has a particular view on the project and uses related tools. Architects use a UML-based tool to analyse and plan the architecture. Developers program the software in Java. These two models overlap: Although they cannot be derived completely from each other, the implementation should follow the architecture, and architects want to see how code changes affect the architecture. So, architects and developers use transformations to translate changes between Java (Java) and UML, as far as automatically possible.

UX designers develop the UI for the software. Their designs overlap with the UML model, because, first, the software's requirements mandate certain properties of the UI, and, second, the architecture may restrict which information can be shown at which point in the interface. The UI design also overlaps with the code, since static parts of the UI can be derived from the UI model. Ideally, changes in the UI code can even be propagated back into the UI model. Thus, transformations are specified between the UI model and UML and between the UI model and Java.

The developers use OpenAPI [The Linux Foundation 2021] to exchange specifications of HTTP APIs. These specifications overlap with the parsing and serialisation code. Thus, developers specify a transformation between Java and **OpenAPI** to keep API specifications consistent with their implementation in code and, for example, to generate Java stubs for added API specifications.

Finally, architects want to analyse how their architecture choices influence performance, using the **Palladio Component Model (PCM)** [Reussner et al. 2016], which provides a simulator for predicting the system's performance based on an annotated architecture specification. The architecture specification used in the PCM overlaps with the one defined in UML. Additionally, the PCM model contains information about performance properties and the deployment structure, which can partially be derived from the code. Thus, architects define transformations between the PCM and Java, as well as between the PCM and UML.

All these transformations avoid re-specification of similar information, such as the architecture in PCM and UML, to derive information, like appropriate Java stubs from OpenAPI (OpenAPI) specifications, and to preserve information consistency. Figure 1 shows the resulting transformation network. In every possible development scenario, different roles may be present and they may use different tools to describe specific kinds of models. Except for rather small projects, having more than one kind of model to describe a system is common, thus automatically preserving their consistency by means of transformations leads to a transformation network. In this article, we discuss how such a network can and needs to be executed, and we find an execution strategy

for such transformations, which is needed to correctly propagate changes from one model to the others.

2.2 Context

We discuss model transformation networks in a specific usage context. We assume that different roles are involved in a development project, each using some models to describe their view of the system. These models inherently overlap, because they all describe the same system, and model transformations are used to keep them consistent. For the sake of simplicity, we only discuss *binary*, bidirectional transformations between two models. They describe how consistency can be restored by updating one model after another was modified.

To foster independent specification and reuse of transformations, we assume that they are not tailor-made but may be general-purpose. For example, a transformation between Java and UML class diagrams may be used across multiple projects and not for one specific software project. As a consequence, we cannot assume that the models or transformations are or can be aligned in any way, for example, to ensure that their execution in a specific order always results in consistent models. Neither can we assume that the network has, or can be modified to have, a certain topology. We do, however, assume that all transformations are in accordance to a well-defined overall notion of consistency (reaching a consistent state would be impossible otherwise). This means, for example, that if three pairwise transformations have been defined between three models, then they cannot specify contradictory constraints, such as naming conventions that cannot be fulfilled at the same time. If in our example scenario depicted in Figure 1 the transformations require for each component in a PCM model the existence of a component in a UML component diagram with the same name as well a class in Java code with the same name, then the transformation between UML and Java may not specify a different naming convention, such as to add a "Component" suffix to the Java class name. Otherwise, these constraints would be contradictory and thus no consistent state could be reached without removing every added component. Such a notion of *compatibility* and an approach to validate it has been proposed in existing work [Klare et al. 2020]. This means that all requirements we pose on the transformations must only concern a transformation itself. A requirement like "no transformation overwrites the result of another" would not fit our context.

Finally, we require that transformations are *synchronising* [Diskin et al. 2016], i.e., that they can deal with the situation that both of their models have been changed. This requirement is essential to find an execution strategy: When propagating changes in a transformation network that contains cycles, it will inevitably happen that both models that are connected by a transformation will be changed. In addition, the well-researched *bidirectional* transformations only change one of the models [Stevens 2010] and could in such a situation be forced to overwrite changes to yield a consistent result. This assumption also enables concurrent modifications of models by different project members.

2.3 Formalisation

We are not concerned with how models are structured, so we simply resort to defining a universe $\mathbb{M}$ that contains all models. This general notion can be applied to any relevant modelling formalism, such as the **Meta Object Facility (MOF)** [Object Management Group (OMG) 2015] and its implementation in the **Eclipse Modeling Framework (EMF)** [Steinberg et al. 2009]. First, we define the kind of transformations that we use:

Definition 1 (Synchronising Transformation). A *synchronising binary transformation (syncx)* $\overrightarrow{t}$ is a function that updates two models:

$$\overrightarrow{t}\colon\ (\mathbb{M} \times \mathbb{M}) \to (\mathbb{M} \times \mathbb{M})$$

A syncx' image consists of fixed points:

$$\forall a \in \mathbb{M}\, \forall b \in \mathbb{M} : \overrightarrow{t}\left(\overrightarrow{t}(a,b)\right) = \overrightarrow{t}(a,b).$$

The universe of all syncx for $\mathbb{M}$ is called $\mathbb{T}$.

This formalisation is a simplification sufficient for the purposes of this article. In practice, transformations will, for example, be specific to certain kinds of models, i.e., for specific *metamodels*, such as the UML, PCM, or Java. And they may be allowed to indicate an error instead of being required to always produce appropriate new models. Additionally, they may process changes instead of model states, because changes can contain information that cannot be recovered by comparing model states [Diskin et al. 2011].

Readers familiar with existing formalisms for bidirectional transformations [Stevens 2010] will observe that there is no explicit consistency relation in the definition of a syncx. For our purposes, the consistency relation is not part of a syncx but rather encoded implicitly in the syncx' behaviour. We assume that transformations are correct and hippocratic [Stevens 2010] with regard to their implicit consistency relation, i.e., they do not change models that are already consistent, and can then recover the relation:

Definition 2 (Consistency Relation). The *consistency relation* $R_{\overrightarrow{t}}$ of syncx $\overrightarrow{t}$ is given by:

$$R_{\overrightarrow{t}} = \left\{(a,b) \mid \overrightarrow{t}(a,b) = (a,b)\right\}.$$

We consider a pair (a,b) *consistent according to* $\overrightarrow{t}$ if and only if $(a,b) \in R_{\overrightarrow{t}}$.

In practice, transformations usually have access to traceability links, also known as *trace links* or *correspondences*, since this is necessary to realise certain notions of consistency [Diskin et al. 2017]. They are used as an auxiliary artefact for a consistency relation, which is why they can, from a theoretical perspective, simply be considered as an additional model in a consistency relation. In consequence, models are only considered consistent with respect to a specific traceability model.

This article focuses on transformation networks that are created when combining multiple syncx:

Definition 3 (Transformation Network). A *transformation network* $N =: ((V,E),T)$ consists of a directed, connected, self-loop-free graph $G = (V,E)$ and a syncx assignment $T : E \to \mathbb{T}$. Any two vertices $\{a,b\} \subseteq V$ have at most one edge between them: $(a,b) \in E \implies (b,a) \notin E$. The universe of all transformation networks for $\mathbb{M}$ is called $\mathbb{U}$.

A transformation network captures the topology and the used transformations. There is no inherent reason to exclude multigraphs or self-loops. We use this simpler definition because it makes it easier to argue about the networks without restricting expressiveness. We use directed edges instead of undirected ones to provide a notion of the "left" and "right" model for a syncx. The edges' direction does not indicate anything about the direction of change propagation. T will usually be injective. Having two assignments of the same syncx would mean that there are two pairs of models that are kept consistent in the same way, which means that either the syncx or the models themselves are redundant. This injectivity is, however, not a formal necessity. We will usually regard the network as given and try to find suitable model assignments:

Definition 4 (Model Assignment). A *model assignment* M for a transformation network $N =: ((V,E),T)$ is a function $M : V \to \mathbb{M}$.

Considering the example in Figure 1, the transformation network only describes the topology of the transformations based on a graph whose nodes represent PCM, UML, OpenAPI, Java, and UI, and whose edges are mapped to the transformations between them, just like depicted in that

graphic. A model assignment then assigns actual models, such as a UML class diagram or Java code, to these nodes. Naturally, we are particularly interested in model assignments that are consistent according to the transformations:

Definition 5 (Consistent Model Assignment). For a transformation network $N =: ((V, E), T)$, a model assignment M is *consistent* if and only if

$$\forall\, (a, b) \in E : (M(a), M(b)) \in R_{T(a,b)}.$$

The set of all consistent model assignments for N is called R_N.

We use the following additional notation in this article:

- "$A \to B$" for the set of functions from set A to set B.
- "f: $A \nrightarrow B$" for a partial function f from A to B.
- "$f(x) = \bot$" to mean that a partial function f is not defined at x.
- "$\mathrm{Im}(f)$" to denote the image of a function f.

2.4 Problem Description

Our goal is to find an algorithm that, given a transformation network $N =: ((V, E), T) \in \mathbb{U}$ and a model assignment M, finds a consistent model assignment M' by applying transformations in $\mathrm{Im}(T)$. We call such an algorithm a *(transformation network) execution strategy*. It is *universal* if it is parameterised by and thus defined for every network.

Definition 6 (Execution Strategy). A universal execution strategy determines an order (i.e., a permutation with duplicates) of transformations in $\mathrm{Im}(T)$ for a given transformation network $N =: ((V, E), T) \in \mathbb{U}$ and model assignment $M \in (V \to \mathbb{M})$. The strategy realises a partial function $S\colon \mathbb{U} \times (V \to \mathbb{M}) \nrightarrow (V \to \mathbb{M})$, whose result is yielded by executing the transformations in the determined order.

An execution strategy finds a new model assignment by only executing the transformations of the network, as more precisely defined by Klare et al. [2021, Definition 8]. It realises a function because the result should be deterministic (though the following considerations and insights also hold if it was non-deterministic and thus only realised a relation), and this function is partial because there may be inputs for which no reasonable result can be achieved by executing the transformations in any order. Considering the example in Figure 1 and a model assignment in which a component was added to the PCM model of a consistent model assignment, executing a universal execution strategy for that transformation network and the given model assignment yields a new model assignment that is consistent again. To achieve that, the strategy may only execute the transformations of the network in an appropriate order, for example, executing the transformation from PCM to UML to add the component to a UML component model, and then the transformations from PCM and UML to Java to add a class that represents the implementation of that component. If necessary, then it may also execute the transformation between Java and OpenAPI if the API needs to be adapted to reflect the Java class. It may even be necessary that transformations have to be executed multiple times to finally achieve consistency of the model assignment according to all transformations.

If $S(N, M) \neq \bot$, then we say that the strategy *resolves* N and M. If $S(N, M) = \bot$, then we say that the strategy fails. We have two further requirements:

Requirement 1 (Correctness). An execution strategy must be correct:

$$\forall N =: ((V, E), T) \in \mathbb{U}\ \forall M \in (V \to \mathbb{M}) : S(N, M) \in R_N \cup \{\bot\}.$$

Requirement 2 (Hippocracticness). An execution strategy must be hippocratic:

$$\forall N =: ((V, E), T) \in \mathbb{U}\ \forall M_c \in R_N : S(N, M_c) = M_c.$$

These two requirements can trivially be satisfied by a strategy that keeps already consistent models unchanged and fails in all other cases. This shows that these are only basic requirements that are also known as essential properties of transformations (see Stevens [2010]), but they do not ensure that a strategy fulfilling them may be considered *useful*. We discuss usefulness of a strategy in the remainder of this section.

An execution strategy will not always be able to find a consistent model assignment (i.e., there will be N, M such that $S(N, M) = \bot$). First, there may not be a consistent model assignment at all (i.e., $R_N = \emptyset$). Second, there may be a consistent model assignment but no execution order of the transformations that yields the assignment [Klare et al. 2019; Stevens 2020b]. We call such inputs *unresolvable* [Stevens 2020b]. Conversely, if an execution order of the transformations that yields a consistent model assignment exists, then we call the inputs *resolvable*. In general, an execution strategy may not terminate in case it does not find an execution order of the transformations that yields consistent models. In this work, we do, however, aim at strategies that terminate always to achieve practical applicability. In particular, in case the strategy fails, we expect it to terminate and indicate the failure.

An execution strategy may even fail for resolvable inputs: The execution strategy may not "find" a consistent model assignment, even though it is reachable. For example, the strategy may abort before having executed the transformations often enough, or finding the assignment might require an order of execution that the strategy does not consider. We will discuss why these situations can occur and may not be avoidable in more detail in Section 4. We call such a strategy *incomplete*:

Definition 7 (Incomplete Execution Strategy). An execution strategy S is *incomplete* if it is correct and if there can be resolvable inputs N, M with $S(N, M) = \bot$.

The higher the probability that an execution strategy yields a result for resolvable inputs (we also say the lower its *level of incompleteness*), the more useful the strategy will be. It is, however, also desirable that the strategy is predictable, meaning that one can determine beforehand for which inputs the strategy will succeed. For example, it would be useful to know whether a strategy yields a result for a given network for *any* resolvable model assignment. Informally speaking, we would like to have an "easy-to-check" criterion for transformation networks determining whether this is the case, which we will define more precisely in Section 5. An even better criterion could be applied to a single syncx, such that the strategy can resolve all inputs with a network of syncx that fulfil the criterion. This would be ideal for the motivated context of independently developing, reusing, and freely combining syncx to a network.

To summarise, we aim to find a correct, hippocratic universal execution strategy that is able to keep models consistent by executing the transformations of transformation networks. The strategy should succeed for realistic inputs with a high probability. Additionally, we aim to find criteria that determine the cases in which the strategy will succeed.

3 RELATED WORK

Approaches for restoring model consistency have been subject to intensive research, surveyed and classified in a feature model by Macedo et al. [2016]. The classification has been extended specifically for the case in which multiple models are to be kept consistent by Stünkel et al. [2021]. Model transformations are a well-researched option, and several tools and languages have been developed to support them [Kusel et al. 2013; Samimi-Dehkordi et al. 2016; Stevens 2008]. Research has, however, mainly focused on consistency between two models. Maintaining

consistency between more than two models has recently gained more attention, especially in terms of a dedicated Dagstuhl seminar [Cleve et al. 2019]. The central approaches of multiary transformations and networks of binary transformations can be distinguished. In Section 1, we have discussed that multiary transformations are complex to specify and limit reusability [Klare 2021], whereas networks of binary transformations have limited expressiveness [Stevens 2020b], which does, however, not seem to be practically relevant [Cleve et al. 2019].

Multiary Transformations. Multiary transformations, i.e., transformations relating multiple (in general more than two) metamodels, have especially been investigated in standards or as extensions of existing approaches or languages for binary transformations. The QVT standard [Object Management Group (OMG) 2016] specifies the relational language QVT-R with support for multidirectionality by design, but ambiguities in the standard limit practical applicability [Macedo et al. 2014]. **Triple Graph Grammars (TGGs)** [Schürr 1995] are bidirectional specifications based on graph patterns, which are well-suited for model transformations [Anjorin et al. 2014]. Extensions of TGGs to multiple models called **Multi Graph Grammars (MGGs)** [Königs and Schürr 2006] and Graph Diagram Grammars [Trollmann and Albayrak 2015, 2016] consider the specification of multidirectional rules. Although a single multiary transformation may be used instead of a complete transformation network, such a transformation has two essential drawbacks. First, it requires the transformation developer to know about and be able to express the relations between all involved metamodels. Second, the more metamodels are related by a multiary transformation, the more limited is its reusability, because it cannot be easily transferred to another project in which a subset of the metamodels together with some other metamodels is used. Thus, a compositional approach for transformations is beneficial. Finally, even if some relations are expressed in multiary transformations, or even need to be expressed in such because of expressiveness reasons [Stevens 2020a], these multiary transformations will or can then be combined with further transformations to a network, such that the work that we present is still required.

Auxiliary Models. Not every multiary relation can be expressed by a set of binary ones. Adding one auxiliary model makes it, however, theoretically possible to express arbitrary multiary relations by binary ones [Stevens 2020b]. Some work discussed which kinds of relations can be expressed with such an approach and how they can be formalised in the lenses framework [Diskin et al. 2018; Stünkel et al. 2018]. Based on that, an approach to use such an auxiliary model to define partial commonalities of models with the same modelling language as used for defining the models themselves has been proposed [Stünkel et al. 2020]. Other work discussed how composing such auxiliary models to express commonalities of models can be achieved [Klare and Gleitze 2019]. Such auxiliary models actually encode a multiary transformation in a model together with binary transformations to the models to keep consistent, resulting in the same challenges as for transformation networks. In consequence, our work on transformation networks is also required and applicable there.

Binary Transformations. Although they cannot express all multiary relations, there are arguments in favour of using networks of modular transformations, especially binary ones: They are easier to develop when domain knowledge is distributed [Klare 2018] and they are easier to comprehend by a single developer [Cleve et al. 2019; Stevens 2020b]. Additionally, binary transformations are researched well, and a variety of tools supporting different kinds of specifying them exist [Kusel et al. 2013; Macedo et al. 2016; Samimi-Dehkordi et al. 2016; Stevens 2008]. Most formalisms and tools consider *bidirectional* transformations, whereas networks require synchronising transformations, as motivated in Section 2.2. There are also approaches focused on processing concurrent changes performed to multiple models, thus also synchronising changes, e.g., Weidmann et al.

[2020]. These approaches, however, need to be able to deal with contradictory changes, which cannot occur within a transformation network due to the assumptions we have made in Section 2.2. Thus, such transformations need to be more powerful but are, for that reason, also more difficult to specify than synchronising transformations. Still such approaches provide a means to achieve synchronising transformations. In addition, non-synchronising transformations can even be adapted to become synchronising [Xiong et al. 2013].

Transformation Chains. Transformation chains combine transformations to derive low-level models from high-level ones across intermediate representations. Languages like FTG+PM [Lúcio et al. 2013] and UniTI [Vanhooff et al. 2007] enable the specification of such chains. Such languages have also been used to define workflows for the tool integration of a development process and apply common model transformation frameworks such as VIATRA [Balogh et al. 2010]. Transformation chains are, however, usually comparably simple, as they represent chains rather than arbitrary network topologies containing cycles. Thus, algorithms for them, in particular for their orchestration, can, in general, not be transferred to networks of arbitrary topology. Etien et al. consider specific properties of transformation chains. They investigate how conflicts in terms of results depending on the execution order can be detected [Etien et al. 2012]. These results, however, do not aim to relieve developers from the task of finding an execution order manually, as we do in this article.

Transformation Composition. Transformation composition techniques are a means to build networks of transformations. They can be separated into internal and external techniques [Wagelaar 2008]. Internal techniques are white-box approaches integrated into the language [Wagelaar et al. 2011], e.g., inheritance or superimposition techniques [Wagelaar et al. 2010]. External approaches consider the transformations as black-boxes. For such transformation compositions, Lano et al. [2014] present a catalogue of patterns that foster correct composition. Our contributions can be seen as an external composition technique. However, composition usually considers transformations between the same rather than different metamodels. From a theoretical perspective (see Section 2.3), this could be treated equally by not distinguishing models by their metamodels. Practical approaches, however, consider transformations between specific metamodels rather than arbitrary models. Bergmann [2021] discusses different problems when composing transformations to networks, e.g., transformations undoing changes. He shows that even if the single transformations operate properly and fulfil common requirements for bidirectional transformations, their composition may not. For that reason, he introduces the property of *very-well behavedness* for multidirectional transformations. Even with that property, orchestration of transformations as discussed in this article is required. However, that property can be considered a reasonable preliminary for the transformations in a network, comparable to the requirement for the transformations to conform to a well-defined overall notion of consistency, as discussed in Section 2.2.

Transformation Termination. Several theoretical properties of transformations have been investigated, upon which *termination* is one of the most important. A couple of termination analyses have been proposed, in particular for graph transformations, which analyse whether the execution of transformation rules terminates [Ehrig et al. 2005]. For graph transformation systems, analyses based on a mapping to Petri nets [Varró et al. 2006] have been proposed. In particular, due to undecidability of the general problem of analysing termination, different termination criteria have been investigated, which have also been combined in a single approach [Bisztray and Heckel 2014]. All these approaches concern termination of a single transformation, more precisely of the execution of the rules of a transformation, whereas our work concerns the combination of multiple transformations and termination thereof.

Execution Strategies. Di Rocco et al. [2017] describe a simple strategy for orchestrating transformations, but make strong assumptions requiring that each of them is only applied once. Stevens [2020b] proposes a strategy that also executes each transformation only once and only in one direction. It includes a notion of *authoritative models*, which are not allowed to be changed, and does not consider synchronising transformations. Likewise, Stevens [2020a] proposes to find an *orientation model* defining in which direction transformations are executed. If, however, several transformations modify the same model, then the approach leaves it to the developer to determine an execution order after which all consistency relations hold. Such strategies are only correct if the network is a tree or if no transformations interfere with each other. We present a simple scenario in which this is already too limiting in Section 4.1. We overcome this limitation by executing transformations more than once and thereby letting them "negotiate" a result even if they interfere, which yields a *universal* execution strategy for arbitrary network topologies.

4 DESIGN SPACE FOR EXECUTION STRATEGIES

We approach the possibilities for designing an execution strategy for transformation networks by looking at how often the strategy executes each transformation in the worst case. The extremes are given by executing every transformation at most once and executing them an unlimited number of times. We will find that neither of them is sufficient: We show that the first one is too limiting to be generally applied and that the second one cannot guarantee termination. As a consequential insight, a universal execution strategy needs to be *incomplete*, introduced as contribution *C2*.

4.1 One Execution per Transformation

Strategies proposed in recent work execute every transformation in a network at most once [Stevens 2020b; Vanhooff et al. 2007]. As motivated in Section 2.2, we expect transformations to be developed independently and thus not to be aligned with each other. Restricting the number of executions to one per transformation would, however, limit the possible combinations of them, and models could not be kept consistent in desirable scenarios. We give an example for this in the following.

We use the example of Section 2.1 and focus on the UML, Java, and OpenAPI models to consider the scenario visualised in Figure 2: Starting with some initial or potentially empty models, an architect creates a new UML interface `ExampleService`. To preserve consistency with the other models by adding the interface to the Java code and the OpenAPI specification, he or she executes the specified transformations. To this end, the architect applies an execution strategy that executes every transformation once. First, the UML-to-Java transformation creates an appropriate interface in Java. The Java-to-OpenAPI transformation recognises that the interface should be exposed via an HTTP API and creates a matching endpoint in the OpenAPI model. Additionally, it creates a stub implementation with parsing and serialisation code in Java. The stub implementation class, however, can not be propagated back to UML, because the UML-to-Java transformation has already been executed, and the execution strategy is assumed to execute every transformation only once.

As a consequential insight, we see that if we limit the number of executions to one per transformation, then transformations cannot propagate back the changes that other transformations have made. However, in the context described in Section 2.2, it is necessary that transformations are able to "react" to the changes made by other transformations. This offers, for instance, separation of concerns, which is necessary for independent development of the single transformations: The logic for a certain aspect of consistency can be put in only one transformation and other transformations will propagate it throughout the network. Without such a mechanism, all aspects of consistency would need to be implemented in all transformations and thus a transformation would be tied to its context, i.e., the other transformations in the network. This would cause duplication

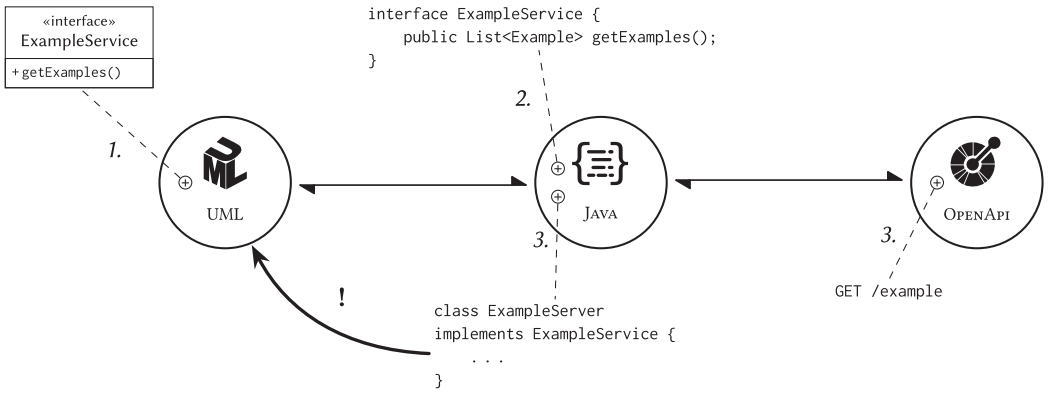

Fig. 2. Example yielding inconsistent models after executing each transformation once. Numbers in italics indicate the order in which changes are performed.

Fig. 3. A transformation network with n transformations reacting to each other.

of logic and reduce reusability of transformations, which would be impractical and contradict our assumption of independent development. More precisely, in the sketched scenario, we would have to add the logic for creating implementations of relevant Java interfaces whenever the interface shall be exposed via an HTTP API to the UML-to-Java transformation. Then, however, the UML-to-Java transformation would implicitly assume the presence of the Java-to-OpenAPI transformation and would be restricted to only be used in networks, and thus in development scenarios, in which OpenAPI is present.

We can generalise the previous example to networks of arbitrary size such that no execution strategy will yield a consistent model assignment after executing each transformation only once: Let the model universe be the natural numbers, i.e., $\mathbb{M} = \mathbb{N}_0$. Let further for every $1 \leq j \leq n$ the syncx $\overleftrightarrow{i_j}$ be defined as follows:

$$\overleftrightarrow{i_j}\colon\ (a,b) \mapsto \begin{cases} (m+1, m+1) & \text{if } m = j \\ (m,m) & \text{else} \end{cases} \qquad \text{with } m \coloneqq \max\{a,b\}\,.$$

$\overleftrightarrow{i_j}$ sets both models to the higher number of the two, except if that number is j. If the higher of the numbers is j, then $\overleftrightarrow{i_j}$ sets both models to $j+1$. This is an abstraction of syncx "reacting" to each other: The $\overleftrightarrow{i_j}$s seek to set all models to the same value, except that after $\overleftrightarrow{i_{j-1}}$ was executed, $\overleftrightarrow{i_j}$ changes its behaviour and increments the value by one.

We now construct the transformation network N_n for $n = 2k, k \in \mathbb{N}^+$ (see Figure 3) with n indicating the number of transformations within the network and examine how many executions it requires to yield a consistent model assignment:

$$T_n = (i, i+1) \mapsto \begin{cases} \overleftrightarrow{i_{2i}} & \text{if } i \leq \frac{n}{2} \\ \overleftrightarrow{i_{2i-n-1}} & \text{else} \end{cases}$$
$$N_n = (([1, n+1], \{(i, i+1) \mid i \in [1, n]\}), T_n)\,.$$

Lemma 1. $\overrightarrow{i_n}$ *must be executed at least n times to resolve* N_n *with the initial model assignment*

$$M_1 : i \mapsto \begin{cases} 1 & \text{if } i = 1 \\ 0 & \text{else.} \end{cases}$$

Proof. The only reachable model assignment that is consistent is $M_n : i \mapsto n$. It is reached by having every $\overrightarrow{i_j}$ increment the highest number in the model assignment by one if that highest number currently is j. All transformations incrementing even numbers are on one side of $\overrightarrow{i_n}$ (except for $\overrightarrow{i_n}$ itself), all transformations incrementing uneven numbers are on the other side. Thus, the currently highest number must be propagated to the other side of $\overrightarrow{i_n}$ at least $n-1$ times. Additionally, $\overrightarrow{i_n}$ must increment $n-1$ to n. In total, $\overrightarrow{i_n}$ must be executed at least n times. □

Theorem 2. *For any execution strategy that uses* $O(1)$ *executions of each transformation, there are inputs that the execution strategy cannot resolve.*

Proof. Follows directly from Lemma 1: The network N_n requires $O(n)$ executions of at least one transformation for the input given in Lemma 1. □

The example network in Figure 2 is a simplification of a realistic transformation scenario as given in Section 2.1 and Figure 2, which we generalised to the network N_n. In consequence of Theorem 2, we can expect that transformation networks can, in general, not be resolved with $O(1)$ executions of each transformation.

4.2 Unlimited Executions

We have considered an execution strategy that executes each transformation only once or even a constant number of times. As an opposite extreme, we now consider an execution strategy that allows an unbounded number of transformation executions. Thus, it executes transformations as long as they still change models and terminates once no more changes occur. This overcomes the shortcoming that we observed with limiting the number of executions to a constant. We will, however, show that we cannot guarantee termination of such an execution strategy. To this end, we prove that it is undecidable whether the strategy will terminate by simulating Turing machines with transformation networks.

Given a Turing machine TM over some alphabet Σ, we construct a transformation network $N_{\text{TM}} =: ((V, E), T_{\text{TM}})$ and a model assignment $M_{\text{TM},x}$ that are resolvable if and only if TM halts on input $x \in \Sigma^*$. In consequence, if we found an execution strategy that terminates for every input and yields consistent models for every resolvable input, then we could use that strategy to solve the halting problem for Turing machines. We assume that TM contains no self-loops as well as no cycles of length 2, i.e., that each transition and each sequence of two transitions changes the state of TM. This is without loss of generality, since duplication and triplication of each state resolves such self-loops and cycles, respectively, (see Klare [2021]). The constructed models consist of a timestamp, the tape content, and the tape position (i.e., $\mathbb{M} = \mathbb{N}_0 \times \Sigma^* \times \mathbb{N}_0$). The network N_{TM} has TM's states as vertices and exactly one directed edge (in arbitrary direction) between each pair of states that have a transition between them. The transformations increment the timestamp, change the tape content, and update the tape position according to TM's transition if and only if the source model's timestamp is higher than the target model's timestamp. Thus, each transformation execution emulates one execution step of TM and stores the state of TM after that execution step in the model changed by the transformation. More formally, let $\text{Tr}(a, b) \subseteq \Sigma \times \{-1, 0, 1\} \times \Sigma$ be the transitions defined between the states a and b (with -1, 0, and 1 indicating the head movements "left," "stay," and "right"). We define T_{TM} with $w|_{p \leftarrow r} := w[0\,..\,p-1] \cdot r \cdot w[p+1\,..\,|w|-1]$ such that a transformation between the two models $\alpha =: (t_a, w_a, p_a)$ and $\beta =: (t_b, w_b, p_b)$ encoding a Turing

machine state with t_a, t_b denoting the timestamps, w_a, w_b denoting the tape contents, and p_a, p_b denoting the tape positions, is defined as follows:

$$\forall\,(a,b)\in E : T_{\text{TM}}\,(a,b)\,(\alpha =: (t_a, w_a, p_a), \beta =: (t_b, w_b, p_b)) \\ = \begin{cases} \left(\alpha, \left(t_a + 1,\ w_a|_{p_a \leftarrow r},\ p_a + d\right)\right) & \text{if } t_a > t_b \wedge \exists\,(w_a\,[p_a], d, r) \in \text{Tr}\,(a,b) \\ \left(\left(t_b + 1,\ w_b|_{p_b \leftarrow r},\ p_b + d\right),\ \beta\right) & \text{if } t_a < t_b \wedge \exists\,(w_b\,[p_b], d, r) \in \text{Tr}\,(b,a) \\ (\alpha, \beta) & \text{else.} \end{cases}$$

Let s be the initial state of TM. We set

$$M_{\text{TM},x} : v \mapsto \begin{cases} (1, x, 0) & \text{if } v = s \\ (0, \varepsilon, 0) & \text{else.} \end{cases}$$

We encode the initial tape content and tape position of TM in the model representing the initial state of TM. Then the network is designed in a way such that there is always only at most one transformation to execute, which emulates the next execution step that TM would have performed. We show this emulating behaviour in the following lemma:

Lemma 3. *Executing the transformations of N_{TM} with initial model assignment $M_{TM,x}$ until no transformations change the model assignment anymore terminates if and only if TM halts on input x. If executing the transformations terminates with the final model assignment M_f, then the model with the highest timestamp in* $\text{Im}(M_i)$ *contains* $\text{TM}(x)$ *as tape content.*

Proof. We can see by induction over the model assignments $M_i,\ i \in \mathbb{N}_0$ created while executing the transformations, i.e., each M_i representing the model assignment at execution time i:

(1) There is exactly one $v \in V$ such that the model $M_i(v) =: (t, x, p)$ has the highest timestamp t of all models in $\text{Im}(M_i)$. This is, by design, always the model that has been changed by the previously executed transformation.
(2) There is at most one edge $(a, b) \in E$ whose transformation is inconsistent, i.e., $(M_i(a), M_i(b)) \notin R_{T_{\text{TM}}(a,b)}$. This follows from the definitions of TM and the last executed transformation. Additionally, $a = v$ or $b = v$, because otherwise there would have been two transformations to which models in $\text{Im}(M_{i-1})$ are inconsistent. We assume without loss of generality $a = v$.
(3) If (a, b) exists, then $m' := M_{i+1}(b)$ will contain the same tape content and the same tape position as would result if TM was executed one step from state v with tape content x and tape position p. This represents the emulating behaviour of TM's transitions by transformations in N_{TM}. Additionally, m' will be the model with the highest timestamp of all models in $\text{Im}(M_{i+1})$.
(4) (a, b) does not exist if and only if TM would halt in state v with tape content x and position p.

In consequence of (1)–(4), if and only if TM halts, there is only a single order of the transformations in N_{TM} in which they can be executed, and its execution yields consistent models. □

Theorem 4. *Let $\mathcal{S}$ be an execution strategy that executes transformations until a consistent model assignment is reached. There are inputs for which it can not be decided whether $\mathcal{S}$ will terminate.*

Proof. Given that we can always decide whether $\mathcal{S}$ will terminate, we could use it to decide the halting problem for a universal Turing machine, as follows from Lemma 3. That problem, however, is undecidable. □

This construction does not only show that there is no universal execution strategy that finds an execution order of transformations that yields consistent models whenever such an order exists. It makes it even unlikely that we can find a practicable criterion that transformations have to fulfil to ensure success of an execution strategy like we have motivated in Section 2.4. Such a criterion should apply to a single transformation, since we want to allow their independent specification and reuse, as motivated in Section 2.2. And such a criterion must, at least, ensure that no transformation network T_{TM} can be constructed out of a Turing machine TM, since an execution strategy would otherwise still not be guaranteed to terminate. The definition of the syncx in $\text{Im}(T_{\text{TM}})$ is, however, structurally simple, such that it is unlikely that a syncx fulfilling the hypothetical criterion would still be apt for most practical use cases.

As another option to avoid undecidability, we could restrict the models' size. The models could then no longer store an unbounded tape and, thus, the transformation network construction could only simulate space-restricted Turing machines. There is, however, no reasonable bound for a *necessary* model size, to which they could be limited. In consequence, determining a universal space bound for models would be a theoretically insufficient, arbitrary, and thus impractical restriction. In practical implementations, it may still be reasonable to specify a model size bound, depending on the usage context to ensure termination in case of unexpected errors, such as implementation faults within transformations. But this would neither solve the theoretic problem nor guarantees to work even in the context it is supposed to be used in as soon as models' size exceeds that bound.

As a different point of view, one could question whether it is relevant if an execution strategy can be guaranteed to terminate. Execution strategies will be used to tell users whether changes they made can be incorporated into the other models automatically. In consequence, users should reliably and timely get a response. We might compare this situation to merging changes in version control systems. There, users also want a reliable and timely response on whether their changes could be incorporated automatically or whether they need to resolve conflicts manually.

Since finding a criterion for transformations to ensure termination of a universal execution strategy seems unlikely due to the given arguments, though we did not show that it is impossible, we have to accept that such an execution strategy may not be successful in every case. For practical applicability, we still want to ensure that its execution terminates, such that we have to define an abortion criterion for the strategy. To this end, we will discuss in the following how such a criterion that leads to useful properties can be defined.

5 A USEFUL EXECUTION STRATEGY

We have shown that without a restricting criterion for transformations, which will likely limit their practical applicability, every universal execution strategy will be *incomplete*: There will be inputs for which it fails, even though there would have been an execution order that leads to a consistent model assignment. In this section, we discuss how to find an appropriate execution order and bound and finally present the *tracing strategy*, constituting contribution *C3*.

5.1 Execution Order: Providing Traceability

The number of transformation executions an execution strategy permits is negatively correlated with its level of incompleteness, because allowing more executions increases the number of tested execution orders and thus increases the chance to achieve a consistent model assignment. In consequence, increasing the number of transformation executions an execution strategy permits lowers its level of incompleteness, i.e., the percentage of cases in which it fails. In contrast, the effects of different orders in which transformations can be executed are not as easy to categorise. The authors developed a model transformation network simulator, which we detail in Section 7. It allows to construct transformation networks and to define execution strategies, which can be applied

step-by-step. All examples presented in this article are also modelled in the simulator. For each examined systematic execution order, such as a depth-first or breadth-first selection, the authors found categories of networks on which the order performed worse than another one in terms of incompleteness. In consequence, the level of incompleteness is not a good sole criterion to evaluate orders by. And even if there was a strategy that reduces the level of incompleteness in contrast to all other strategies, there are further properties that are even more important for practical applicability, which we discuss in the following.

We know that a universal execution strategy will inevitably be incomplete, i.e., it will possibly fail for resolvable inputs. In practice, it will be important how well an execution strategy provides traceability in such cases, i.e., how well it helps users to understand where and why the strategy failed with the selected execution order. Such reasons can, for example, be errors within a single transformation, errors in their interplay, or a too early abortion of the execution. The order plays a decisive role in this regard, which is why we focus on finding a strategy that improves the order. Imagine, for instance, that the strategy executed transformations in an arbitrary order until some limit is reached. Users might then be confronted with a situation where all transformations have been executed, but the last model assignment is only consistent according to some of them. There would be no clear pattern and little clues for users where to start investigating the failure's cause. To improve traceability, the authors thus propose the following principle for an execution order, which is based on incrementally increasing the size of the subnetwork of already executed transformations:

Principle 1 (Incremental Subnetwork Consistency). Ensure consistency among the transformations that have already been executed before executing a transformation that has not been executed yet.

Since a syncx can change both models, executing it may result in models that are inconsistent according to the syncx that have been executed previously. We have seen this behaviour in the exemplary scenario sketched in Figure 2 and explained in Section 4.1: The addition of a UML interface is propagated to a Java interface by the UML-to-Java transformation, to which the models are consistent afterwards. This interface is then transformed to an HTTP API in OpenAPI, for which, in turn, a stub implementation for serialisation is created in Java. Both these changes are performed by the Java-to-OpenAPI transformation, as a syncx is allowed to change both models. Then, the models are again inconsistent according to the UML-to-Java transformation because of the stub implementation class being only present in Java code but not in the UML model. Following Principle 1, inconsistencies with already executed transformations should be addressed first. Thus, if there is also a Java-to-PCM transformation, like in the motivating example in Figure 1, then this transformation would only be executed after consistency to both other transformations has been restored. In effect, a strategy applying the principle will maintain a subnetwork of syncx with a consistent model assignment and try to expand the subnetwork transformation by transformation.

To exemplify how Principle 1 provides *traceability*, suppose that an execution strategy applying that principle fails after having executed the set of syncx $E \subseteq \mathbb{T}$. Let $\overrightarrow{t} \in E$ be the last syncx that was executed for its first time. This means that consistency according to all $\overrightarrow{t'} \in E \setminus \{\overrightarrow{t}\}$ has been restored before $\overrightarrow{t}$ was executed, but after executing $\overrightarrow{t}$ no execution order of the syncx was found that restores consistency according to all $\overrightarrow{t'} \in E$. The strategy can then inform users that integrating $\overrightarrow{t}$ into the subnetwork induced by E failed. Furthermore, it can inform users that a result that is consistent according to the syncx in $E \setminus \{\overrightarrow{t}\}$ exists. By that, users gain valuable information for handling the error: First, when trying to understand the error, they can ignore any syncx that is not in E. Second, some aspect of consistency that is present in the consistency relation realised by $\overrightarrow{t}$ but absent in the consistency relations realised by the syncx in $E \setminus \{\overrightarrow{t}\}$ hinders the strategy from

creating a consistent result. Third, when users try to find a consistent model assignment manually, they can start with the consistent result that exists for $E \setminus \{\overrightarrow{t}\}$ instead of having to start from scratch.

5.2 Execution Bound: Reacting to Each Other

Due to Theorem 2, we need to restrict the number of transformation executions with a function in $\omega(m)$ with m being the number of syncx in the input network. Such a limit should be reasonable to support most practical use cases and is thus a tradeoff: Not allowing enough transformation executions reduces the usefulness of the strategy, since not all relevant networks can be resolved. Allowing too many executions might make the strategy run for a long time before aborting, without adding much value.

In Section 4.1, we have motivated that syncx should be able to "react" to each other. We have seen that this excludes any bound in $O(1)$ for the number of executions per transformation, but to guarantee termination, we can also not allow transformations to react to each other indefinitely. If a syncx $\overrightarrow{t}$ changes the models and the other already executed syncx have reacted to those changes by adapting the models to be consistent according to them as well, then $\overrightarrow{t}$ should not react by changing the models again. Because if $\overrightarrow{t}$ changed the models again, then this could easily result in executing the same sequences of transformations repeatedly and there would likely be no consistent result.

We call transformations in a network N that behave in the described way *N-converging*. This is not a property of a transformation on its own but relative to its network N. Thus, it cannot be achieved just by proper construction of the individual transformations and, unfortunately, there is also no simple way to check it statically. Nevertheless, it captures the sensible expectation for transformations explained above. We induce an execution bound for an execution strategy by only requiring it not to fail if all syncx are N-converging. We will see how this execution bound behaves in combination with Principle 1 in the subsequently presented execution strategy.

Definition 8 (N-convergence). Let $N =: (G, T)$ be a transformation network. A syncx $\overrightarrow{t} \in \mathrm{Im}(T)$ is *N-converging* if for every initial model assignment and each subset of the syncx $T_p \subseteq \mathrm{Im}(T)$ with $\overrightarrow{t} \in T_p$ the resulting model assignment is consistent according to $\overrightarrow{t}$ whenever $\overrightarrow{t}$ has been executed after a sequence of the syncx in T_p that contains each permutation of those syncx as a (not necessarily continuous) subsequence.

We only require that the sequence of transformation executions contains each permutation, but allow other executions in between. As an example, assume a network N of N-converging syncx $\overrightarrow{t_1}$, $\overrightarrow{t_2}$, and $\overrightarrow{t_3}$. After executing them in the order $\overrightarrow{t_1}\ \overrightarrow{t_2}\ \overrightarrow{t_3}\ \overrightarrow{t_1}\ \overrightarrow{t_2}\ \overrightarrow{t_3}$, the current model assignment may still be inconsistent according to $\overrightarrow{t_1}$, because $\overrightarrow{t_1}$ was not executed after the order $\overrightarrow{t_3}\ \overrightarrow{t_2}$. After executing $\overrightarrow{t_1}$ once more, the resulting model assignment must now be consistent according to all syncx: $\overrightarrow{t_1}$ was executed after the two orders of other syncx $\overrightarrow{t_2}\ \overrightarrow{t_3}$ and $\overrightarrow{t_3}\ \overrightarrow{t_2}$. Likewise, $\overrightarrow{t_2}$ was executed after $\overrightarrow{t_1}\ \overrightarrow{t_3}$ and $\overrightarrow{t_3}\ \overrightarrow{t_1}$, and $\overrightarrow{t_3}$ was executed after $\overrightarrow{t_1}\ \overrightarrow{t_2}$ and $\overrightarrow{t_2}\ \overrightarrow{t_1}$.

5.3 The Tracing Strategy

We derive a concrete strategy that realises the discussed design choices, namely, Principle 1 and guaranteed termination when fulfilling Definition 8. Algorithm 1 gives pseudocode for such a strategy, which we call the *tracing strategy*. At a high level, it acts like this: Given a changed model assignment, the strategy picks the next `candidate` syncx to execute. After executing the `candidate`, the strategy calls itself on the `subnetwork` formed by the already `executed` syncx. By that, it propagates the changes of the last execution throughout the `subnetwork` and ensures that they are consistent according to the `executed` syncx. Finally, the strategy executes the initial `candidate` again to ensure that the changes added during the subnetwork propagation are consistent according to the `candidate`. If that repeated execution of the `candidate` generates new changes

ALGORITHM 1. The tracing strategy in pseudocode.

```
1  Procedure propagate(network, changes):
2      executed ← ∅
3      accumulatedChanges ← changes
4      Invariant: accumulatedChanges applied to network consistent to executed
5      while network.select(candidate | candidate ∉ executed ∧ accumulatedChanges.adjacentTo(candidate)) do
6          candidateChanges ← candidate.execute(accumulatedChanges)
7          subnetwork ← network.edgeInducedSubgraph(executed)
8          propagationChanges ← propagate(subnetwork, accumulatedChanges ∪ candidateChanges)
9          candidateChanges ← candidate.execute(propagationChanges)
10         if candidateChanges.adjacentToAny(executed) then
               // Only happens if candidate is not network-converging
11             fail(executed, propagationChanges)
12         accumulatedChanges ← propagationChanges ∪ candidateChanges
13         executed ← executed ∪ candidate
14     return accumulatedChanges
```

in any model that is kept consistent by an already executed syncx, then the execution fails, because the `candidate` does not fulfil the definition of being *N-converging*, as we will see in the following. In that case, the procedure returns the already `executed` syncx to which consistency was restored by the also returned changes to support a user in examining the reasons for the strategy to fail. We will discuss this benefit in more detail in Section 6. If the models are consistent according to the `candidate`, then the strategy picks the next one, while a further `candidate` exists. In effect, the strategy realises Principle 1 in a recursive fashion and ensures that each permutation of all yet executed syncx is executed at every recursion level.

Figure 4 depicts an exemplary execution of the strategy for a network with four models and four transformations. We assume that after an initially consistent state of the models, the topmost one was modified. The strategy then picks the top left, the top right, the middle, and the bottom left transformation, and after executing each of them in that order, it recursively executes itself on the `subnetwork` of the particular already `executed` transformations. Since each recursion only treats the `subnetwork` of previously executed transformations, the `network` gets smaller at each recursion level.

Unlike the formalisation in Section 2.3, the algorithm is based on changes instead of model states. Changes contain information that cannot be recovered by comparing model states [Diskin et al. 2011] and are processed by change-driven transformations (see Bergmann et al. [2012]). Thus, in practice, we want to support change-based execution. Since changes just provide further information compared to only having changed model states but the latter can still be derived by applying the former, the previous formal considerations still hold for transformations that process changes. The algorithm also uses changes to determine potential candidates for the next transformation to execute: It only picks candidates that are adjacent to a model that was changed. The input `changes` describe all changes that occurred since the last model assignment M that was known to be consistent. The procedure returns `accumulatedChanges` that, when applied to M, yield a new model assignment M'. For our formalisation, M' is the algorithm's output.

To realise an execution strategy according to Definition 6, the algorithm has to be deterministic. To achieve that, in particular the selection of a candidate in Line 5 has to be implemented deterministically. This can, for example, be achieved by predefining an order of the transformations

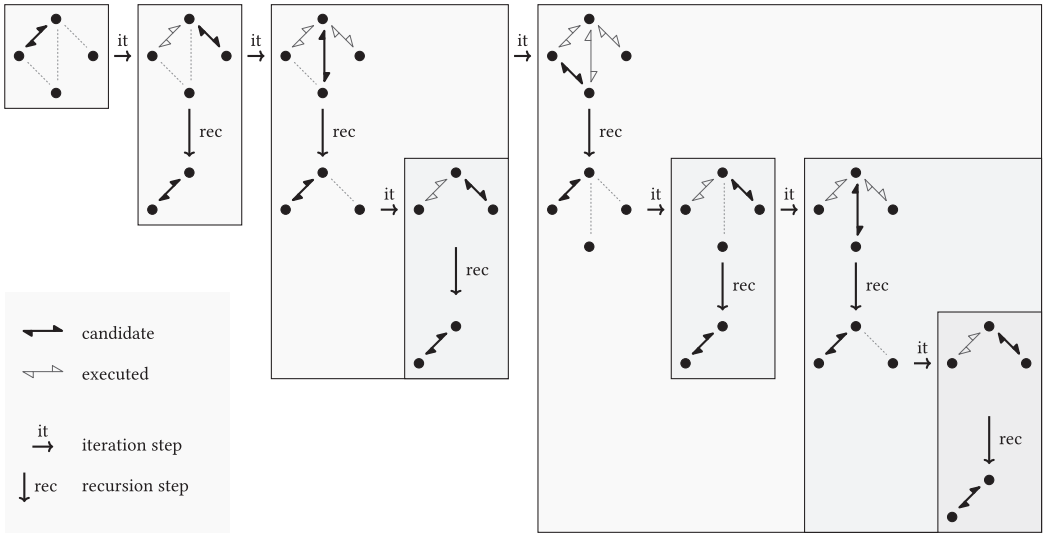

Fig. 4. Exemplary execution of the tracing strategy for a change in the topmost model, depicting the iterations (horizontal) and recursion steps (vertical).

in a network so the selection yields the same transformations in every execution of the algorithm for the same inputs. Although we consider an execution strategy as deterministic in the definition, we have stated in Section 2.4 that this is not even a necessary, but may only be a reasonable requirement, such that it would also be valid if the proposed strategy operated non-deterministically.

We discuss some implementation details for the tracing strategy further below. First, we prove that the strategy has indeed the motivated properties. We assert that it terminates always, determine its execution bound, and show its correctness (Requirement 1) and hippocraticness (Requirement 2), as required in Section 2.4. In addition, we prove that the strategy fulfils the motivated Principle 1 and that it is successful when the transformations fulfil Definition 8 of being N-converging.

Theorem 5. *The tracing strategy terminates for every input.*

Proof. Because all called functions terminate, only the loop (Line 5) and the recursive call in Line 8 can lead to non-termination. Let m denote the number of edges of the network N. The set `executed` is initialised to be empty (Line 2) and grows by one element in every iteration of the loop. The loop is executed no more than m times, because after m iterations there is no transformation that is not in `executed` and, thus, the loop condition cannot be fulfilled.

The recursive call receives a network that is smaller than network N in terms of the number of edges, because it does not contain the current `candidate`. If network N is empty, then the algorithm will not enter the loop and not make a recursive call. Hence, the recursive stack never gets higher than m.

In total, both the number of recursive calls as well as the number of loop iterations are limited by the number of transformations in the network, such that the tracing strategy terminates for every input. □

Theorem 6. *The tracing strategy executes transformations at most $O(2^m)$ times.*

Proof. Let $T(m)$ denote the number of transformation executions the algorithm invokes for a network N with m edges. The set `executed` is initialised to be empty and grows by one

transformation every loop iteration (Line 13). It follows that the recursive call in Line 8 receives a network that is one transformation larger each time. Thus, we find:

$$T(0) = 0$$

$$T(m) = 2m + \sum_{i=0}^{m-1} T(i) = 2 + 2\,T(m-1) = 2\,(2^m - 1) \in O(2^m)\,. \qquad \square$$

Next, we show that the strategy fulfils the fundamental Requirements 1 and 2 regarding correctness and hippocraticness, which we defined in Section 2.4.

Theorem 7. *The tracing strategy is correct.*

Proof. We assume the contrary, i.e., that the strategy produces a model assignment M for network N such that $M \notin R_N$. That means there is an edge $(a, b) \in E$ such that $(M(a), M(b)) \notin R_{\overrightarrow{t}}$, where $\overrightarrow{t} := T(a, b)$. We distinguish these three cases:

(1) $\overrightarrow{t}$ *was never executed:* Then `accumulatedChanges` never contained any change adjacent to a or b (Line 5). Since the initial `changes` were relative to a consistent model assignment, the model assignments of a and b were consistent as well, i.e., we know that $(M(a), M(b)) \in R_{\overrightarrow{t}}$.
(2) $\overrightarrow{t}$ *was executed and no other transformation adjacent to* a *or* b *was executed afterwards:* Then the execution of $\overrightarrow{t}$ yields consistent models, such that $(M(a), M(b)) \in R_{\overrightarrow{t}}$ per definition.
(3) $\overrightarrow{t}$ *was executed and another transformation* $\overrightarrow{u}$ *adjacent to* a *or* b *was executed afterwards:* Because $\overrightarrow{u}$ was executed after $\overrightarrow{t}$, $\overrightarrow{t}$ was in `executed` when $\overrightarrow{u}$ was the `candidate`. So $\overrightarrow{t}$'s last execution was in the recursion after $\overrightarrow{u}$'s first execution in Line 6. Afterwards, $\overrightarrow{u}$ was only executed in Line 9. If $\overrightarrow{u}$ would have changed $M(a)$ or $M(b)$, then the strategy would have raised a failure. Hence, $M(a)$ and $M(b)$ are the same as after the execution of $\overrightarrow{t}$, and $(M(a), M(b)) \in R_{\overrightarrow{t}}$.

All cases lead to a contradiction. $\square$

Theorem 8. *The tracing strategy is hippocratic.*

Proof. The strategy only produces changes by executing transformations, which, per definition, only generate changes if the models are not in their consistency relations. $\square$

Finally, we verify that we have indeed realised Principle 1 and that the strategy does not fail for a network N of only N-converging transformations according to Definition 8.

Theorem 9. *The tracing strategy ensures consistency among the transformations that have already been executed before executing a transformation that has not been executed yet (see Principle 1).*

Proof. After the recursive call in Line 8, the current model assignment is consistent according to all `executed` transformations (Theorem 7) and no changes to models adjacent to an executed transformation are allowed after the recursive call. Hence, `executed` is either empty or the current model assignment is consistent according to all syncx in `executed` whenever the algorithm executes a new transformation in Line 6. $\square$

To prove that the tracing strategy always succeeds if the syncx of the input network N fulfil Definition 8 of being N-converging, we first show a lemma about the order in which transformations have been executed after running the strategy.

Lemma 10. *After running the tracing strategy for an input network* N *of only* N*-converging syncx, the sequence of executed syncx contains each permutation of those syncx (not necessarily continuously).*

Proof. We show the lemma by induction over the number m of edges in network N. First, we note that when calling the algorithm on a network N with m transformations, the first $m-1$ iterations of the loop act identically to executing the algorithm on a network without the last `candidate`. Second, we note that the second part of the loop condition, "`accumulatedChanges.`*adjacentTo*`(candidate)`" (Line 5), does not change the algorithm's result apart from controlling the order in which the syncx are executed. If any syncx was never executed because of this condition, then executing it would not have changed any model. Hence, we assume without loss of generality that all syncx in network N will get executed by Algorithm 1.

The lemma statement is trivially true for $m = 1$. Assume that the statement is true for all networks of size $1 \leq n < m$ but not true for a network of size m. That means after executing the last iteration of the loop, there is an order o of the m syncx in network N in which they have not been executed yet. Let $\overrightarrow{t}$ be the `candidate` of the last iteration. Let j be the index of $\overrightarrow{t}$ in o. Per induction assumption, the order $o[1] \ldots o[j-1]$ has been executed in the previous iterations of the loop. Afterwards, $\overrightarrow{t}$ was executed in Line 6. Per induction assumption, the order $o[j+1] \ldots o[m]$ has been executed in the recursive call (Line 8) of the last iteration. This happened after Line 6. Hence, the transformations have been executed in the order o. This is a contradiction. □

Theorem 11. *If the input network N of the tracing strategy consists only of N-converging syncx, then the tracing strategy does not fail.*

Proof. Calling the algorithm on a network N with m transformations, the first $m-1$ iterations of the loop act identically to executing the algorithm on a network without the last `candidate` (see proof for Lemma 10). Thus, when reaching Line 10 all permutations of transformations in `executed` $\cup$ {`candidate`} have been executed according to Lemma 10. This is even the case when reaching Line 9, because `candidate` has already been executed in Line 6 after the previous loop iteration has executed all permutations in `executed`, such that all permutations of transformations in `executed` $\cup$ {`candidate`} have been executed before Line 9. Then due to Definition 8 for N-convergence, the models are consistent according to all transformations in `executed` $\cup$ {`candidate`} when reaching Line 9, such that `candidateChanges` produced in Line 9 will be empty and the condition leading to a failure (Line 10) will never evaluate to *true*. □

The tracing strategy only guarantees to yield a consistent model assignment if all transformations are N-converging. Unfortunately, we cannot provide an approach to achieve N-convergence by construction or to validate N-convergence for a network of transformations. We have, however, also discussed that every universal execution strategy needs to be incomplete and will thus fail in certain cases. In consequence, even if a network N contains transformations that are not N-converging, the tracing strategy is still incomplete and at least fails based on the notion of a sensible and well-defined property rather than in an arbitrary state. In addition, following Principle 1 eases finding the reasons why the strategy fails, as we will discuss in more detail in Section 6.

The exponential worst-case performance of the strategy is no limitation and does only represent a bound to ensure termination. In cases in which the strategy terminates, we expect the repeated execution of each transformation to perform only few changes in reaction to the changes made by other transformations, as otherwise their execution will unlikely converge to a consistent model assignment, i.e., the transformations are unlikely to be N-converging.

In its current formulation, the tracing strategy does not prevent the transformations from overwriting the initial user changes. This seems inappropriate: The transformations could simply revert the user changes to produce a consistent model assignment, which may usually not be the expected behaviour. Other authors address this issue by forbidding changes to models that have been edited by users [Di Rocco et al. 2017; Stevens 2020a, b]. Stevens calls these models *authoritative* [Stevens

2020b]. However, there are practical use cases where changes to models that were edited by the users should be allowed—the example we presented in Section 4.1 is one of them. An option would be to lock only the changes made by users instead. As soon as a transformation makes a change that would effectively overwrite the users' changes, the execution strategy raises an error. This is a kind of monotony requirement, which, however, limits practical applicability [Klare 2021, pp. 212].

We might argue that users sometimes make changes that rely on a certain unchanged part of the model. If the transformations overwrite that part, then the user changes are contradicted without overwriting the changes themselves. Such a situation might, however, indicate that there is a relevant aspect of the desired consistency relation that the transformations have not taken into account. Then, the transformations need to be improved rather than the execution strategy changed.

The presented tracing strategy provides a universal execution strategy for transformation networks, which ensures termination by being incomplete and realises Principle 1 as an inductive approach for restoring consistency by incrementally adding transformations for which consistency is restored. We assume this approach to improve traceability of the reasons whenever the strategy fails to find consistent models, which we will discuss in more detail and for which we will provide evidence in the following section.

6 REASONING WITH THE TRACING STRATEGY

In Section 5, we have proposed the tracing strategy as an execution strategy for model transformation networks. The strategy is proven correct, i.e., it terminates for every input and, if not failing, the returned models are always consistent. We have motivated the strategy with its ability to improve traceability in cases in which it fails, i.e., in which it does not yield consistent models. This can be the case for two reasons: First, an execution order of the transformations that yields consistent models may exist but the strategy cannot find it because of undecidability reasons (see Theorem 2). Second, there may not even be an execution order of the transformations of a network for given inputs that yields consistent models.

In this section, we aim to show that the tracing strategy actually helps transformation developers to find the cause for the execution strategy not being able to execute the transformations in an order that yields consistent models, introduced as contribution *C4*. To do so, we quantitatively examine the number of transformations a developer has to consider in such a situation and qualitatively analyse how he or she is supported in finding the relevant transformations to investigate. We discuss the improvement in traceability by the proposed strategy at a general example for the first of the two reasons. From this scenario, we derive a universal, systematic criterion for how the strategy improves traceability, which we substantiate with a metric. In a practical scenario aligned with the one depicted in Figure 1, we then generalise how the criterion and the improvement in traceability also applies in an example for the second failure reason. This reason is that no execution order of the transformations that yields consistent models exists due to some kind of incompatibility between the transformations. The ideas and examples in this section are partially based on those presented in the dissertation of Klare [2021].

6.1 Traceability at an Example

According to the discussion in Section 5, we expect the tracing strategy as specified in Algorithm 1 to improve traceability. Traceability is, in particular, improved by reducing the number of transformations the developer or user of a transformation network has to consider when the strategy fails to deliver consistent models. This is achieved by systematically adding transformations to the execution incrementally after consistency to the already executed transformations has been

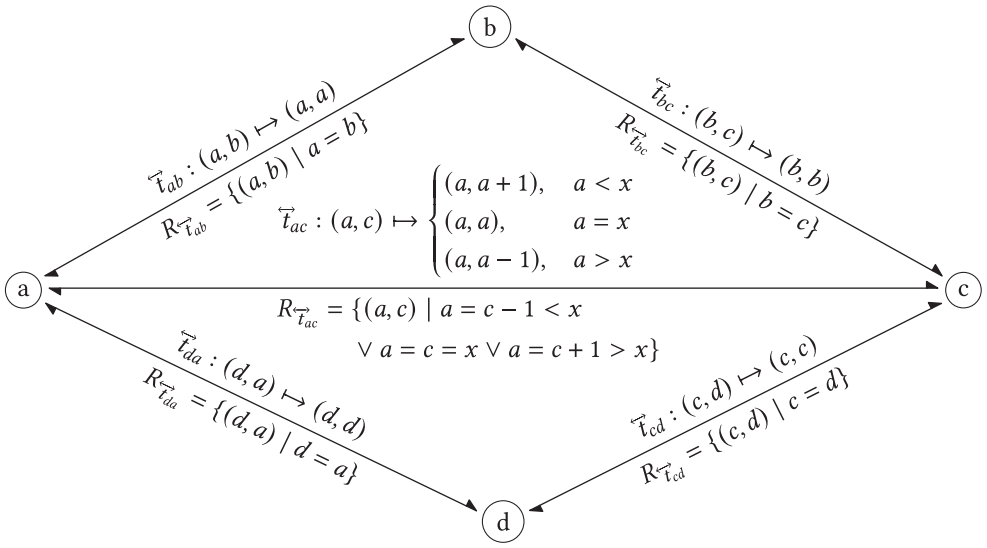

Fig. 5. A network of transformations that can require an arbitrary number of executions to find consistent models, depending on the value x and the input models. Each of the models a, b, c, and d represents a single number, i.e., $a, b, c, d \in \mathbb{N}_0$.

restored, as defined in Principle 1. In this section, we give a comprehensible impression of how this kind of traceability is achieved by discussion at an example.

Figure 5 depicts an abstract scenario with a transformation network consisting of five transformations that preserve consistency between four models. For reasons of simplicity, we only consider models in this section and disregard the abstraction of *model assignments* that assign actual models to the nodes of an abstract graph describing the transformation network topology according to Definitions 3 and 4. The universe of the four models a, b, c, and d is the natural numbers, i.e., $\mathbb{M} = \mathbb{N}_0$. Four of the transformations, namely, $\vec{t}_{\text{ab}}$, $\vec{t}_{\text{bc}}$, $\vec{t}_{\text{cd}}$, and $\vec{t}_{\text{da}}$, set the number of both models to the value of one of the models, such that both have the same value. Thus, their induced consistency relations are the pairs of models being the same number. The transformation $\vec{t}_{\text{ac}}$ sets the value of the model c to the value of the model a incremented by 1 if $a < x$, to the value a decremented by 1 if $a > x$, and to the value of a if $a = x$ for some fixed but arbitrary x.

As a consequence, the only models that are consistent according to all induced consistency relations are $a = b = c = d = x$. For any initial value of a, these consistent models can, for example, be found by executing the sequence $\vec{t}_{\text{ac}}$, $\vec{t}_{\text{cd}}$, $\vec{t}_{\text{da}}$ for $|a-x|+1$ times, such that $a = c = d = x$, followed by $\vec{t}_{\text{ab}}$ and $\vec{t}_{\text{bc}}$, such that $b = x$ as well. This is sufficient, because $\vec{t}_{\text{ac}}$ increments or decrements the models' value in each iteration by 1, and the other transformations propagate this value back to a. Thus, an execution order of the transformations that yields consistent models exists. Since x is an arbitrary value, an arbitrary high number of executions of $\vec{t}_{\text{ac}}$ can be required to find consistent models. To ensure termination of an execution strategy, the number of executions it performs has to be limited, as also implemented in the proposed tracing strategy. Thus, any useful execution strategy applied to the scenario will not find resolutions for several inputs, although they are resolvable (see Section 2.4).

An execution strategy with an artificial termination criterion will fail in an unexpected state, in which the models may be arbitrarily inconsistent, and without any guarantee for the usefulness of the state in which it fails to identify the reason for the failure. The tracing strategy fails under

more systematic conditions due to the subsequent addition of transformations to be executed, providing more insights into the reasons for failing. One such condition is that the execution closes a cycle of transformations that do not interoperate properly. In the example, this situation is given by the transformations $\overrightarrow{t}_{ac}$, $\overrightarrow{t}_{ab}$, and $\overrightarrow{t}_{bc}$ whenever the initial value $a \neq x$: There is no execution order of the transformations that yields models consistent to them. The only consistent models are $a = b = c = x$ but none of the transformations is able to change a or b. Thus, as soon as the execution of any of the three transformations closes a cycle, i.e., as soon as a transformation becomes executed such that the graph induced by the already executed transformations, in this case $\overrightarrow{t}_{ac}$, $\overrightarrow{t}_{ab}$, and $\overrightarrow{t}_{bc}$, contains a cycle, the algorithm fails. Since consistency to previously executed transformations could be preserved, there is a fault in any of the transformations in the cycle closed by the last executed transformation, i.e., `candidate`, or in their interaction. It is, however, impossible to say which transformation contains a fault, because it is unclear whether the consistency relation induced by $\overrightarrow{t}_{ac}$ is actually not as intended and should thus be adapted or whether, for example, $\overrightarrow{t}_{ab}$ and $\overrightarrow{t}_{bc}$ should be adapted. This is a semantic decision by the developer of the transformations.

There are further reasons for the algorithm to fail beyond closing cycles. We will discuss in the following under which conditions we can actually expect closing cycles to be the *only* case in which the algorithm fails. If the algorithm can only fail when the last `candidate` transformation closes a cycle with the `executed` transformations, then the transformations a user or developer has to consider to find the cause for failing are systematically limited: As soon as the algorithm fails, the user or developer receives the information that transformation was executed last and led to the failure together with the current state of the models. There is at least one consistency relation that is violated such that the algorithm aborted, and this relation must belong to one of the transformations within the cycle containing the fault. In consequence, the transformation user or developer only needs to consider the transformations in that cycle for finding the fault. He or she knows which consistency relation was violated and can thus restrict him- or herself to the elements concerned with the violated consistency relation. In this case, both cycles that may be closed by adding $\overrightarrow{t}_{ac}$ (the one with $\overrightarrow{t}_{ab}$ and $\overrightarrow{t}_{bc}$ as well as the one with $\overrightarrow{t}_{da}$ and $\overrightarrow{t}_{cd}$) are of length 3, such that independent from the order in which the tracing strategy executes the transformations, only 3 out of the 5 transformations must be considered when the algorithm fails. In general, the number of transformations to consider is limited by the closed cycle's length. Using an execution strategy that may fail in an arbitrary state, there is no clue which transformations caused the failure.

6.2 Possible Execution States in Failure Cases

Although the previous considerations suggest that only closing cycles in the graph induced by the `executed` transformations leads to failures of an execution strategy, this is not necessarily the case. Consider again the scenario from Figure 5. We discuss the execution of the tracing strategy for a transformation network with $x > 1$ and initial models $a = 1$, $b = 3$, and arbitrary values of the others. Assume that the strategy starts executing $\overrightarrow{t}_{bc}$ in the first iteration (Line 5), then afterwards $c = b = 3$, which is the initial value of b. Then assume that the strategy selects and executes $\overrightarrow{t}_{ac}$ in the second loop iteration, such that afterwards $c = a + 1 = 2$. Then the recursive execution of the strategy in Line 8 executes the `executed` transformations again, which is $\overrightarrow{t}_{bc}$ and which resets c to $c = b = 3$ again, because $b \neq c$ and thus the models are not consistent according to $\overrightarrow{t}_{bc}$. After returning from the recursion in Line 9, $\overrightarrow{t}_{ac}$ is executed again and resets c to $c = a + 1 = 2$, such that a change adjacent to $\overrightarrow{t}_{bc}$ is generated and the validation in Line 10 leads to a failure of the algorithm in Line 11. An iterative execution of only $\overrightarrow{t}_{ac}$ and $\overrightarrow{t}_{bc}$ will never terminate, because the execution of one transformation always leads to an inconsistency according to the other. In consequence, failing in this case is intended behaviour of an execution strategy.

In the discussed scenario, one might argue that the transformations are somehow incompatible, because the execution of the two mentioned transformations will never result in models consistent to both of them. There is, however, always an execution order of all transformations in the network that terminates, i.e., every input is resolvable.

Generalising from the example, there is no necessity for closing a cycle in the network induced by the `candidate` and the `executed` transformations for the tracing strategy to fail, because a transformation does not consider which model was changed and updates the other model accordingly, but it only takes the models' states and updates both of them. In the scenario, model c is always changed by both transformations, although the transformations are executed because of a change of c. Updating the model that was changed and led to an execution of the transformation can, however, always be necessary, even if the other model is updated as well, as we have motivated with Figure 2. In consequence, every iteration in the tracing strategy adding the execution of a further transformation can result in a failure (Line 11), no matter whether it closes a cycle or not.

6.3 A Criterion for Improved Traceability

The tracing strategy can only fail if some model is changed multiple times, because when failing some models need to be inconsistent (the ones connected by the last executed `candidate` transformation) as detected in Line 10, for which consistency was already restored when first executing the `candidate` transformation in Line 6, such that another transformation must have changed one of the models again. This can occur because of a cycle in the executed transformations, like in the first discussed scenario in Section 6.1, or because of transformations that also modify the model whose previous change introduced an inconsistency, such as $\overleftrightarrow{t}_{bc}$ updating c even if only c was changed before, like in the second discussed scenario in Section 6.2.

In practice, transformations may also consider which models have been changed, thus not only considering the models' state but also the changes since their last consistent state, as an extension of Definition 1 (see Klare et al. [2021]). Such a transformation can decide in which direction an update has to be performed and avoid behaviour like in $\overleftrightarrow{t}_{bc}$, which always reverts changes of c even if b was not changed. If a network consists of transformations that after a change of one model only update the other one, then only closing cycles can lead to a failure, because only then the changed model is adjacent to a transformation that has already been executed and thus leads to other transformations being executed. In that case, it is beneficial to first close cycles by choosing the next transformation within the algorithm in Line 5 to ensure that it fails as early as possible. Even if transformations in practice will not always only modify one model after the other was changed but may also update the originally changed model, as motivated for the behaviour of a transformation between Java and OpenAPI in Figure 2, we may expect that in many cases an update of one model is sufficient to restore consistency, as also defined in common notions of bidirectional transformations, e.g., Stevens [2010]. Then, the aforementioned strategy of identifying the cause for the algorithm to fail by investigating only the transformations in the cycle of the `executed` transformations closed by the last `candidate` applies.

We have shown that the first execution of a transformation can lead to a failure in rather different situations, such as the discussed scenario of closing a cycle but also the simple case of executing only two adjacent transformations. From these insights, we can construct situations in which the tracing strategy only fails after executing all transformations. This may even be due to a specific execution order. For example, for $x = 2$ and $a = b = c = d = 0$, the tracing strategy fails when choosing the execution order $\overrightarrow{t}_{ab}$, $\overrightarrow{t}_{bc}$, $\overrightarrow{t}_{cd}$, $\overrightarrow{t}_{da}$, $\overrightarrow{t}_{ac}$ only after each of the five transformations was executed. Still, we can derive a systematic criterion for the expected number of transformations that need to be considered when the tracing strategy fails for a specific transformation network

and actual model assignments, and we can show why it is beneficial compared to other execution strategies.

Usually, identifying the cause of a failure of the transformations' execution will not require all transformations to be investigated but only a specific subset of them. This may actually not only depend on the subset of k of the m transformations in the network but also on their execution order, like we have seen in the second scenario as discussed in Section 6.2, such that the execution of these k transformations only fails if they are executed in a specific order. With an arbitrary execution strategy, it is unclear when such a problematic order of transformations is executed. In particular, this may happen after arbitrary other, unrelated transformations have been executed, which can, in the worst case, be all other transformations in the network. Any strategy that executes further transformations although consistency according to the already executed transformations has not yet been achieved is prone to executing many or even all transformations before failing, since it is unclear when a possible termination criterion applies and restricts the number of already executed transformations. In consequence, the expected value for the size of the subset of transformations being executed before the strategy fails will be near m with such a strategy.

In contrast, the tracing strategy fails as soon as each of the k transformations has been executed, because due to Principle 1 these transformations then have been executed in every possible order, including the problematic order. In the worst case, the selection of transformations in Line 5 happens in a completely random way. Then the probability that the rth of the k problematic transformations has been executed after executing any e of the m transformations in the network, denoted as event X, is given by the negative hypergeometric distribution. We are interested in the case that the last of the k transformations has been executed, such that $r = k$, which gives us the probability:

$$P(X = e) = \frac{\binom{e-1}{r-1} \times \binom{m-e}{k-r}}{\binom{m}{k}} = \frac{\binom{e-1}{k-1}}{\binom{m}{k}}.$$

The expected number of transformations that need to be executed for the tracing strategy to fail because of the k problematic transformations is then given by the expected value of X:

$$E(X) = \sum_{e=k}^{m} e \times \frac{\binom{e-1}{k-1}}{\binom{m}{k}} = \frac{m+1}{k+1} \times k.$$

In consequence, using the tracing strategy with a random selection of the next `candidate` transformation to execute, we can expect the strategy to fail after $\frac{m+1}{k+1} \times k$ of the m transformations have been executed when there is a problematic execution order of k of the transformations like we have seen in the previous example. Applied to the scenario in Figure 5, we can expect the tracing strategy to fail on average after executing $\frac{6}{3} \times 2 = 4$ transformations, because the two transformations $\overleftarrow{t}_{\text{bc}}$ and $\overleftarrow{t}_{\text{ac}}$ are problematic when executed one after another.

The tracing strategy thus systematically reduces the number of transformations a developer needs to consider whenever the strategy fails by the realisation of Principle 1. In practice, we may even improve this by a reasonable selection of the next `candidate` transformation to execute, such as a transformation closing a cycle rather than executing a transformation to a further model that has not been changed before. We have discussed before why we expect closing cycles to be prone to produce failures and under which conditions that is the only case in which the algorithm can fail.

As a further improvement, we may execute the tracing strategy multiple times and investigate only the pass in which the lowest number of transformations was executed before the algorithm failed. This can even further improve traceability by reducing the number of transformations to consider. Although this approach may also be applied to any other execution strategy, it will be

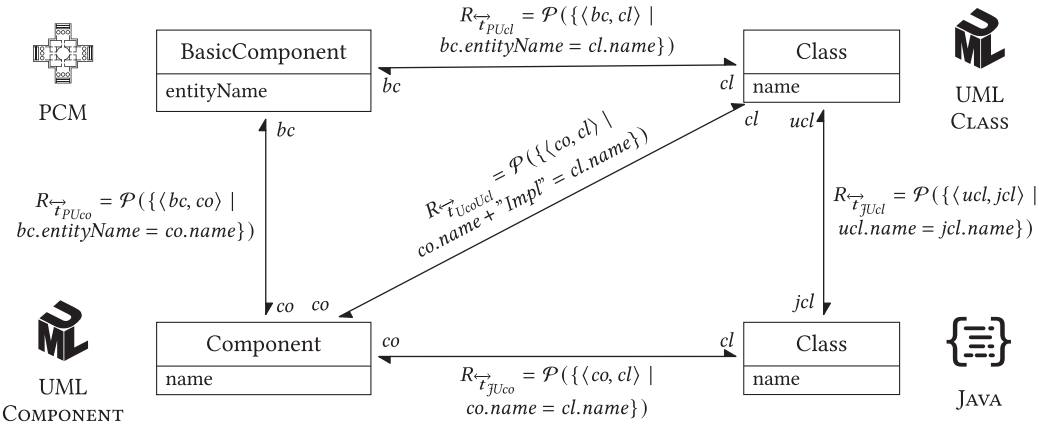

Fig. 6. Consistency relations between basic components in PCM models, components in UML component models, classes in UML class models, and classes in Java models.

more beneficial when executing the tracing strategy because of its lower expected value for the number of transformations to be executed until a failure occurs.

6.4 Generalisation to Incompatible Transformations

In Section 2.2, we have discussed that we assume transformations to be in accordance with some well-defined overall notion of consistency, such that there are no *incompatibilities* between the transformations that prevent them from finding consistent models. A formal notion of compatibility has been introduced in Klare et al. [2020], which is based on the idea that for every model element whose consistency is constrained by a transformation, there must be a consistent set of models containing that element. This ensures that the consistency relations induced by the transformations do not contradict each other such that they can never be fulfilled.

Figure 6 depicts a simplified scenario from component-based software engineering according to the one in Figure 1 using different representations of software components and their realising classes. Components represented in a PCM model and in a UML component model are kept consistent with each other and with a representation as classes in UML class diagrams and Java code. All model elements are reduced to their names for reasons of simplicity. The example contains consistency relations, which are defined by the power sets of pairs of components and classes such that all models are considered consistent when they contain the same number of classes and components, respectively, that fulfil some constraint regarding their names. These constraints require the existence of classes and components with equal names, except for the constraint between UML components and UML classes, which requires an "Impl" suffix to be appended to the class name, according to the pattern proposed by Langhammer [2017]. We consider these relations *incompatible*, because (except for the empty sets) no models fulfil the defined constraints and are thus consistent according to these relations. This also conforms to the notion of incompatibility defined in Klare et al. [2020].

Since there are no consistent models according to these relations, any transformations that preserve consistency according to the relations will not be able to yield consistent models (despite returning the empty models). Each of the transformations may add or remove the "Impl" suffix or repeatedly add elements with a further "Impl" suffix to locally restore consistency, such that the models infinitely alternate between the same states or diverge with new states containing an

increasing and, if not aborting, infinite number of elements. Thus, every input will be unresolvable and, in particular, every execution strategy will fail.

However, the independent development and modular combination and reuse of single transformations can easily lead to a transformation network that contains such incompatible transformations. The situation depicted in Figure 6 can easily occur if different people, potentially with different roles, develop the individual transformations or reuse them from other projects, since these people or projects rely on slightly different notions of when models shall be considered consistent. These notions are then reflected in the consistency relations or the transformations that induce them. Even worse, compatibility is, in general, undecidable and can only be proven in specific situations and for specific kinds of relations [Klare et al. 2020], such that when combining transformations their compatibility can be validated in many cases, but is usually unknown.

The discussed improvements in traceability through the tracing strategy even apply in case the transformations are incompatible, because the reason why inputs are unresolvable, be it a problematic behaviour of the transformations, such as in the example in Figure 5, or incompatibilities in the underlying or induced relations, does not influence the behaviour of the tracing strategy. As soon as a cycle including $\overrightarrow{t}_{\text{UcoUcl}}$ is closed during the execution of the tracing strategy, i.e., as soon as the subnetwork induced by the `executed` transformations contains a cycle including $\overrightarrow{t}_{\text{UcoUcl}}$, the strategy will fail. This is because the three relations of each of the two simple cycles in the transformation network including $\overrightarrow{t}_{\text{UcoUcl}}$ are already incompatible and thus the transformations will not yield consistent models for them. According to the criterion introduced in Section 6.3, we expect the tracing strategy to fail on average after $\frac{6}{4} \times 3 = 4,5$ transformations have been executed, thus restricting the number of transformations a developer has to consider to find the cause of the failure accordingly.

6.5 Discussion and Validity

The scenarios and the derived criterion give us useful and systematic insights about the usefulness of the tracing strategy in terms of its ability to improve traceability after a failure. In the following, we summarise these insights and discuss the validity of our discussion and its implications.

6.5.1 Insights. We have seen at different execution scenarios for the transformation network depicted in Figure 5 that the tracing strategy can fail in different situations, such as closing a cycle within the already executed transformations but also the execution of a chain of (potentially even only two) transformations. We have argued that under specific assumptions these kinds of situations can be restricted. For example, having transformations that after a change of one model only update the other one, closing a cycle within the already executed transformations is the only situation in which the strategy can fail. In other cases, independent from the actual network topology, the strategy may fail after having executed all transformations.

Nevertheless, in general, if an execution strategy fails, then this will usually already be the case for a subset of the transformations in the transformation network. Thus, even for a subset of the transformations, no execution order can be found that yields consistent models. This is why and where the tracing strategy provides an essential benefit by the implementation of Principle 1 that ensures that the set of transformations to which consistency has been restored is increased incrementally. In consequence, as soon as the subset of transformations, for which no execution order that restored consistency can be found, has been executed, the tracing strategy fails. We found that for a network containing n transformations with a subset of $k \leq n$ transformations for which no execution order that yields consistent models exists, the tracing strategy will fail on average after executing $\frac{n+1}{k+1} \times k$ transformations. This systematically reduces the number of transformations to consider in case of a failure compared to other execution strategies, which have

no systematic approach of selecting the executed transformations, such that on average most of the transformations will have been executed when such a strategy fails.

As a further insight, we can improve the strategy's behaviour in case we know that transformations do, at least in most cases, only update one model after the other was changed and do not change both or even only the one that has been modified before. In those cases, we know that the tracing strategy can only fail if cycles are closed in the subnetwork induced by the already executed transformations, such that first closing cycles is beneficial to provoke early failures. As an example, consider the scenario in Figure 6. If we first execute $\overrightarrow{t}_{\text{PUco}}$ and $\overrightarrow{t}_{\text{PUcl}}$, then it is beneficial to then execute $\overrightarrow{t}_{\text{UcoUcl}}$ to close a cycle, as the tracing strategy then already fails. Executing the transformations to Java first delays the failure of the algorithm. Thus, the selection of the next `candidate` in Line 5 should consider transformations to which changes of both related models have already been performed first.

6.5.2 Validity. The tracing strategy systematically limits the expected number of transformations that need to be considered when it fails to find an execution order of the transformations that yields consistent models. We expect this to improve efficiency in finding the cause of such a failure, because traceability improves when fewer transformations need to be considered. There are, however, threats to the validity of this conclusion that need to be considered.

First, we assume that the lower the number of transformations to consider after a failure, the easier it is to find the causing fault. Although this relation is a sensible expectation, as considering unrelated transformations will likely increase complexity and required time for the process of identifying the cause of a failure, this is a potential threat to validity. To mitigate this threat, we did not only focus on the metric regarding the expected number of transformations to be considered, but we also presented qualitative arguments and discussed further quantifiable improvements, such as possible restrictions when the transformations fulfil specific requirements regarding which models they may change upon execution.

Second, we have compared the proposed tracing strategy with possible other execution strategies for transformation networks that do not ensure consistency according to already executed transformations before executing further ones. In this comparison, we can expect an improvement in the average-case number of transformations to consider in case of a failure. There may, however, be another strategy that performs better or at least equal to the proposed one in all cases, which is a threat to external validity of the results. We have mitigated this issue by systematically deriving the strategy based on the well-defined Principle 1, which systematically improves the investigated kind of traceability. Beyond traceability, we have developed a simulator (as mentioned in Section 5 and to be detailed in Section 7) for evaluating different execution strategies, but we found each strategy to be outperformed by at least one other strategy regarding its ability to find an execution order that yields consistent models in at least one scenario. Thus, we do not expect another strategy to be systematically better than the one we proposed, but, in the best case, only to perform better in specific situations in terms of the ability to find an execution order that yields consistent models.

Both kinds of validity can be improved by empirical evaluation, which is why we plan to perform a controlled experiment in which the information delivered by the tracing strategy and by other strategies are presented to different groups of developers. Evaluating how long they take to find and fix faults, e.g., in terms of the time they take or the number of steps they need, and how successful they are in both situations helps us to validate the expected improvement in traceability, i.e., the efficiency in finding the cause of a failure, and improve evidence of our conclusions from the given scenario-based discussion. Additionally, qualitative statements from interviews can be evaluated. Such an experiment requires high effort in terms of its conduction and evaluation, but

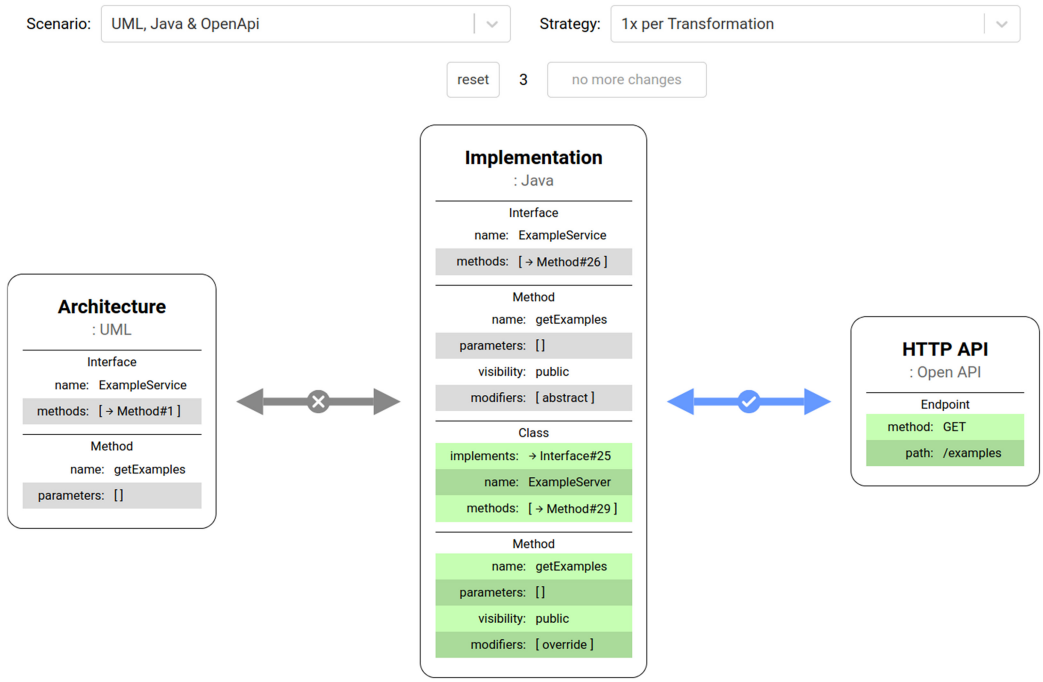

Fig. 7. Screenshot of executing the UML, Java, and OpenAPI example (see Figure 2) in the simulator.

also because of the required presence of adequate transformation networks of proper size and complexity.

7 A SIMULATOR FOR EXECUTION STRATEGIES

During their research on execution strategies, the authors noticed that evaluating and comparing how different strategies behave is difficult. Since the goal was to find a universal strategy, many representative examples had to be considered. Once models and transformations are more complex than the simplest of examples, it becomes challenging to keep track of which changes the transformations apply to which models.

To address this challenge, the authors developed a model transformation network simulator, introduced as contribution *C5*. It allows to construct transformation networks and to define execution strategies. A strategy can be applied step-by-step on a network. The simulator gives a graphical representation of the network's state and allows users to follow along as the strategy executes. This is exemplarily shown in the screenshot in Figure 7, which shows an execution step of the transformation network as depicted in Figure 2. The graphical representation helps to provide an understanding of how a strategy behaves. One can construct examples for which different strategies yield different results and thereby find their boundaries.

The simulator uses a simple metamodelling mechanism and change representation. Compared to full-fledged modelling frameworks, it makes it easier to define models and transformations. It is not based on a heavyweight framework such as EMF with all its dependencies and even a specific IDE, but can be used in plain Java-based languages and thus facilitates rapid prototyping and easy reproducibility. Nevertheless, the simulator supports all features the authors consider relevant to simulate realistic scenarios: model objects with attributes and references; change-based transformations; change types for model object addition, deletion, and feature modification; and

traceability links. The simulator is realised as a web application and programmed in Kotlin. It is available online [Gleitze 2020a] with the source code at GitHub [Gleitze 2020b]. All examples presented in these papers are also modelled in the simulator and can there be tried out interactively.

The authors used the simulator, for example, to compare different orders for execution strategies regarding their level of incompleteness. The simulator helped to come to the conclusion that each of them is inferior to others in at least one situation, as discussed in Section 5. The simulator's benefit is that it enables researchers and developers to produce comparable, reproducible, and comprehensible realisations of different strategies with low effort. It is, thus, supposed to help moving general research on transformation network execution strategies forward. Any contributions to the simulator, be it models, transformations, strategies, or technical improvements, are most welcome.

8 CONCLUSION AND FUTURE WORK

Transformations can be used to keep multiple models describing a single software system consistent. Developing them independently and reusing them across different projects requires the possibility to dynamically compose them to a network. A universal execution strategy then needs to decide in which order these transformations have to be executed to restore consistency of all models according to every transformation.

In this article, we have discussed influencing factors for designing such a universal execution strategy for model transformation networks. It involves determining an order to execute the transformations in and a bound for the number of executions. We have proven that every universal execution strategy that guarantees termination needs to be incomplete, i.e., it will possibly fail for certain cases in which an execution order of transformations that yields a consistent solution exists. We have argued that providing traceability in cases where an execution strategy fails should be a central design goal. As a result, we have proposed the *tracing strategy*, which is proven correct and terminates for every input. Additionally, we have shown that this strategy has a well-defined bound for the number of transformation executions to ensure a reasonable level of incompleteness and that it actually improves traceability of failures in case the strategy does not yield consistent models. Finally, we have presented a simulator for transformation networks, which is publicly available and can be used to recapitulate the scenarios presented in this article but also to evaluate further execution strategies at different scenarios.

Our findings on execution bounds and the behaviour of the proposed execution strategy have been provided in a formalised way to prove the insights and expected properties of the strategy. In consequence, this article provides fundamental knowledge about the design space and relevant design goals of transformation network execution strategies. While the statements on correctness and well-definedness are proven, those on the usefulness of the strategy in terms of improving traceability were derived by argumentation. By construction, the number of transformations to consider in case the algorithm fails is restricted by the number of already executed transformations. In addition, we have provided evidence for improving traceability by deriving from example scenarios that the execution usually already fails for a subset of the transformations, and then the expected number of executed transformations when executing the complete network until a failure occurs is limited by the size of this subset of transformations.

In future work, the authors want to examine how the strategy can be further optimised: It might, for example, be improved by backtracking and by selecting the next candidate transformation more carefully, taking the network topology and nature of the changes into account. Since early executed transformations will be executed most often, it might be beneficial to start with those that are most likely not to cause conflicts. Etien et al. present heuristics for identifying such transformations [Etien et al. 2010, 2012], which might prove valuable in this regard. In addition, this

article only discussed networks of binary transformations. The presented execution strategy, however, does not rely on the transformations being binary and may work just as well for networks with multiary transformations. Future research could investigate whether there are relevant differences when applying the execution strategy to networks of multiary transformations. Finally, the tracing strategy is proven successful as long as the transformations fulfil the definition of being N-converging, which is a sensible notion for the allowed interaction of transformations. This property can, however, neither be easily guaranteed nor analysed. We have argued why this is still a reasonable property, but a property that can at least be analysed at design time to avoid failures during execution would be interesting for theoretic considerations, even if it impractically restricts expressiveness of transformations.

The authors also plan to provide further evidence of practical applicability of the proposed strategy in terms of the usefulness of its provided properties in a controlled experiment. In such an experiment, the strategy is compared to others by letting different groups of developers apply them to multiple scenarios and investigating their required effort to identify the cause of failures. Beyond applicability, the authors plan to evaluate performance of the strategy, in particular by showing that the worst-case performance in $O(2^m)$ is actually not a practical problem. To this end, they implement the strategy in the Vitruvius framework [Klare et al. 2021] for consistent system development, in which transformations are currently orchestrated by a simple depth-first selection strategy. Additionally, they plan to provide an extension of the established benchmark framework for bidirectional transformations *BenchmarX* [Anjorin et al. 2020] to the multi-model case for both the validation of the strategy proposed in this article as well as the possibility to compare different transformation approaches and different execution strategies based on a common benchmark.

VERIFIABILITY

We provide the artefacts of the transformation network simulator (see Section 7), namely, the sources and a generated ready-to-use web application, in a dedicated reproduction package [Klare and Gleitze 2022]. In this simulator, the tracing strategy presented in this article as well as other execution strategies for comparison are implemented and can be executed on different example scenarios, containing the one based on UML, Java, and OpenAPI that we have depicted in Figure 2.

ACKNOWLEDGMENTS

We thank Erik Burger for supporting our work as a reviewing and editing author for the initial conference version of this article. We also thank all the reviewers for their valuable, in-depth feedback to this article.

REFERENCES

Anthony Anjorin, Thomas Buchmann, Bernhard Westfechtel, Zinovy Diskin, Hsiang-Shang Ko, Romina Eramo, Georg Hinkel, Leila Samimi-Dehkordi, and Albert Zündorf. 2020. Benchmarking bidirectional transformations: Theory, implementation, application, and assessment. *Softw. Syst. Model.* 19, 3 (5 2020), 647–691. DOI: https://doi.org/10.1007/s10270-019-00752-x

Anthony Anjorin, Sebastian Rose, Frederik Deckwerth, and Andy Schürr. 2014. Efficient model synchronization with view triple graph grammars. In *10th European Conference on Modelling Foundations and Applications (ECMFA)*, Jordi Cabot and Julia Rubin (Eds.). Springer International Publishing, 1–17. DOI: https://doi.org/10.1007/978-3-319-09195-2_1

András Balogh, Gábor Bergmann, György Csertán, László Gönczy, Ákos Horváth, István Majzik, András Pataricza, Balázs Polgár, István Ráth, Dániel Varró, and Gergely Varró. 2010. *Workflow-driven Tool Integration Using Model Transformations.* Springer-Verlag, Berlin, 224–248. DOI: https://doi.org/10.1007/978-3-642-17322-6_11

Gábor Bergmann. 2021. Controllable and decomposable multidirectional synchronizations. *Softw. Syst. Model.* 20, 5 (2021), 1735–1774. DOI: https://doi.org/10.1007/s10270-021-00879-w

Gábor Bergmann, István Ráth, Gergely Varró, and Dániel Varró. 2012. Change-driven model transformations. *Softw. Syst. Model.* 11, 3 (2012), 431–461. DOI: https://doi.org/10.1007/s10270-011-0197-9

Dénes Bisztray and Reiko Heckel. 2014. Combining termination proofs in model transformation systems. *Math. Struct. Comput. Sci.* 24, 4 (Aug. 2014), 240407. DOI: https://doi.org/10.1017/S0960129512000369

Jean-Michel Bruel, Benoit Combemale, Esther Guerra, Jean-Marc Jézéquel, Jörg Kienzle, Juan de Lara, Gunter Mussbacher, Eugene Syriani, and Hans Vangheluwe. 2020. Comparing and classifying model transformation reuse approaches across metamodels. *Softw. Syst. Model.* 19, 2 (2020), 441–465. DOI: https://doi.org/10.1007/s10270-019-00762-9

Anthony Cleve, Ekkart Kindler, Perdita Stevens, and Vadim Zaytsev. 2019. Multidirectional transformations and synchronisations (Dagstuhl Seminar 18491). *Dagst. Rep.* 8, 12 (2019), 1–48. DOI: https://doi.org/10.4230/DagRep.8.12.1

Juri Di Rocco, Davide Di Ruscio, Marcel Heinz, Ludovico Iovino, Ralf Lämmel, and Alfonso Pierantonio. 2017. Consistency recovery in interactive modeling. In *3rd International Workshop on Executable Modeling (EXE) co-located with ACM/IEEE 20th International Conference on Model Driven Engineering Languages and Systems (MODELS)*. CEUR-WS.org, 116–122. Retrieved from http://ceur-ws.org/Vol-2019/exe_6.pdf.

Zinovy Diskin, Hamid Gholizadeh, Arif Wider, and Krzysztof Czarnecki. 2016. A three-dimensional taxonomy for bidirectional model synchronization. *J. Syst. Softw.* 111 (2016), 298–322. DOI: https://doi.org/10.1016/j.jss.2015.06.003

Zinovy Diskin, Abel Gómez, and Jordi Cabot. 2017. Traceability mappings as a fundamental instrument in model transformations. In *20th International Conference on Fundamental Approaches to Software Engineering (FASE)*. Springer, Berlin, 247–263. DOI: https://doi.org/10.1007/978-3-662-54494-5_14

Zinovy Diskin, Harald König, and Mark Lawford. 2018. Multiple model synchronization with multiary delta lenses. In *21st International Conference on Fundamental Approaches to Software Engineering (FASE)*. Springer International Publishing, 21–37. DOI: https://doi.org/10.1007/978-3-319-89363-1_2

Zinovy Diskin, Yingfei Xiong, Krzysztof Czarnecki, Hartmut Ehrig, Frank Hermann, and Fernando Orejas. 2011. From state- to delta-based bidirectional model transformations: The symmetric case. In *14th International Conference on Model Driven Engineering Languages and Systems (MODELS)*. Springer, Berlin, 304–318. DOI: https://doi.org/10.1007/978-3-642-24485-8_22

Hartmut Ehrig, Karsten Ehrig, Juan de Lara, Gabriele Taentzer, Dániel Varró, and Szilvia Varró-Gyapay. 2005. Termination criteria for model transformation. In *8th International Conference on Fundamental Approaches to Software Engineering (FASE)*, Maura Cerioli (Ed.). Springer, Berlin, 49–63. DOI: https://doi.org/10.1007/978-3-540-31984-9_5

Anne Etien, Vincent Aranega, Xavier Blanc, and Richard F. Paige. 2012. Chaining model transformations. In *1st Workshop on the Analysis of Model Transformations (AMT)*. ACM, 9–14. DOI: https://doi.org/10.1145/2432497.2432500

Anne Etien, Alexis Muller, Thomas Legrand, and Xavier Blanc. 2010. Combining independent model transformations. In *ACM Symposium on Applied Computing (SAC)*. ACM, 2237–2243. DOI: https://doi.org/10.1145/1774088.1774557

Joshua Gleitze. 2020a. Transformation Network Simulator (Application). Retrieved from https://jgleitz.github.io/transformationnetwork-simulator.

Joshua Gleitze. 2020b. Transformation Network Simulator (GitHub Repository). Retrieved from https://github.com/jGleitz/transformationnetwork-simulator.

Joshua Gleitze, Heiko Klare, and Erik Burger. 2021. Finding a universal execution strategy for model transformation networks. In *24th International Conference on Fundamental Approaches to Software Engineering (FASE)*, Esther Guerra and Mariëlle Stoelinga (Eds.). Springer International Publishing, 87–107. DOI: https://doi.org/10.1007/978-3-030-71500-7_5

Houssem Guissouma, Heiko Klare, Eric Sax, and Erik Burger. 2018. An empirical study on the current and future challenges of automotive software release and configuration management. In *44th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*. IEEE, 298–305. DOI: https://doi.org/10.1109/SEAA.2018.00056

Heiko Klare. 2018. Multi-model consistency preservation. In *21st ACM/IEEE International Conference on Model Driven Engineering Languages and Systems: Companion Proceedings (MODELS)*. ACM, 156–161. DOI: https://doi.org/10.1145/3270112.3275335

Heiko Klare. 2021. *Building Transformation Networks for Consistent Evolution of Interrelated Models*. Ph.D. Dissertation. Karlsruhe Institute of Technology, Karlsruhe, Germany. DOI: https://doi.org/10.5445/IR/1000133724

Heiko Klare and Joshua Gleitze. 2019. Commonalities for preserving consistency of multiple models. In *22nd ACM/IEEE International Conference on Model Driven Engineering Languages and Systems Companion (MODELS)*. IEEE, 371–378. DOI: https://doi.org/10.1109/MODELS-C.2019.00058

Heiko Klare and Joshua Gleitze. 2022. Reproduction Package for the Paper on Termination and Expressiveness of Execution Strategies for Networks of Bidirectional Model Transformations. DOI: https://doi.org/10.5445/IR/1000145276

Heiko Klare, Max E. Kramer, Michael Langhammer, Dominik Werle, Erik Burger, and Ralf Reussner. 2021. Enabling consistency in view-based system development – The Vitruvius approach. *J. Syst. Softw.* 171 (2021). DOI: https://doi.org/10.1016/j.jss.2020.110815

Heiko Klare, Aurélien Pepin, Erik Burger, and Ralf Reussner. 2020. *A Formal Approach to Prove Compatibility in Transformation Networks*. Technical Report 3. Karlsruhe Institut für Technologie (KIT), Karlsruhe. DOI: https://doi.org/10.5445/IR/1000121444

Heiko Klare, Torsten Syma, Erik Burger, and Ralf Reussner. 2019. A categorization of interoperability issues in networks of transformations. *J. Object Technol.* 18, 3 (2019), 4:1–20. DOI: https://doi.org/10.5381/jot.2019.18.3.a4

Alexander Königs and Andy Schürr. 2006. MDI: A rule-based multi-document and tool integration approach. *Softw. Syst. Model.* 5, 4 (2006), 349–368. DOI: https://doi.org/10.1007/s10270-006-0016-x

Angelika Kusel, Juergen Etzlstorfer, Elisabeth Kapsammer, Philip Langer, Werner Retschitzegger, Johannes Schoenboeck, Wieland Schwinger, and Manuel Wimmer. 2013. A survey on incremental model transformation approaches. In *Workshop on Models and Evolution (ME) co-located with 6th ACM/IEEE International Conference on Model Driven Engineering Languages and Systems (MODELS)*. CEUR-WS.org, 4–13. Retrieved from http://ceur-ws.org/Vol-1090/1.pdf.

Michael Langhammer. 2017. *Automated Coevolution of Source Code and Software Architecture Models.* Ph.D. Dissertation. Karlsruhe Institute of Technology, Karlsruhe, Germany. DOI: https://doi.org/10.5445/IR/1000069366

Kevin Lano, Shekoufeh Kolahdouz-Rahimi, Poernomo Iman, Jeffrey Terrell, and Steffen Zschaler. 2014. Correct-by-construction synthesis of model transformations using transformation patterns. *Softw. Syst. Model.* 13, 2 (2014), 873–907. DOI: https://doi.org/10.1007/s10270-012-0291-7

Levi Lúcio, Sadaf Mustafiz, Joachim Denil, Hans Vangheluwe, and Maris Jukss. 2013. FTG+PM: An integrated framework for investigating model transformation chains. In *16th International SDL Forum: Model-driven Dependability Engineering (DSL)*. Springer, Berlin, 182–202. DOI: https://doi.org/10.1007/978-3-642-38911-5_11

Nuno Macedo, Alcino Cunha, and Hugo Pacheco. 2014. Towards a framework for multi-directional model transformations. In *3rd International Workshop on Bidirectional Transformations (BX) (Vol-1133)*. CEUR-WS.org. Retrieved from http://ceur-ws.org/Vol-1133/paper-11.pdf.

Nuno Macedo, Tiago Jorge, and Alcino Cunha. 2016. A feature-based classification of model repair approaches. *IEEE Trans. Softw. Eng.* 43, 7 (2016), 615–640. DOI: https://doi.org/10.1109/TSE.2016.2620145

Object Management Group (OMG). 2015. Meta Object Facility (MOF). Retrieved from http://www.omg.org/spec/MOF/2.5.

Object Management Group (OMG). 2016. Meta Object Facility (MOF) 2.0–Query/View/Transformation Specification. Retrieved from http://www.omg.org/spec/QVT/1.3.

Ralf H. Reussner, Steffen Becker, Jens Happe, Robert Heinrich, Anne Koziolek, Heiko Koziolek, Max Kramer, and Klaus Krogmann. 2016. *Modeling and Simulating Software Architectures – The Palladio Approach.* The MIT Press. Retrieved from http://mitpress.mit.edu/books/modeling-and-simulating-software-architectures.

L. Samimi-Dehkordi, B. Zamani, and S. Kolahdouz-Rahimi. 2016. Bidirectional model transformation approaches – A comparative study. In *6th International Conference on Computer and Knowledge Engineering (ICCKE)*. IEEE, 314–320. DOI: https://doi.org/10.1109/ICCKE.2016.7802159

Andy Schürr. 1995. Specification of graph translators with triple graph grammars. In *20th International Workshop on Graph-Theoretic Concepts in Computer Science (WG)*. Springer, Berlin, 151–163. DOI: https://doi.org/10.1007/3-540-59071-4_45

David Steinberg, Frank Budinsky, Marcelo Paternostro, and Ed Merks. 2009. *EMF: Eclipse Modeling Framework* (2nd ed.). Addison-Wesley Professional. Retrieved from https://dl.acm.org/doi/book/10.5555/1197540.

Perdita Stevens. 2008. A landscape of bidirectional model transformations. In *International Summer School on Generative and Transformational Techniques in Software Engineering II (GTTSE)*. Springer, Berlin, 408–424. DOI: https://doi.org/10.1007/978-3-540-88643-3_10

Perdita Stevens. 2010. Bidirectional model transformations in QVT: Semantic issues and open questions. *Softw. Syst. Model.* 9, 1 (2010), 7. DOI: https://doi.org/10.1007/s10270-008-0109-9

Perdita Stevens. 2020. Maintaining consistency in networks of models: Bidirectional transformations in the large. *Softw. Syst. Model.* 19, 1 (2020), 39–65. DOI: https://doi.org/10.1007/s10270-019-00736-x

Perdita Stevens. 2020. Connecting software build with maintaining consistency between models: Towards sound, optimal, and flexible building from megamodels. *Softw. Syst. Model.* 19, 4 (2020), 935–958. DOI: https://doi.org/10.1007/s10270-020-00788-4

Patrick Stünkel, Harald König, Yngve Lamo, and Adrian Rutle. 2020. Towards multiple model synchronization with comprehensive systems. In *23rd International Conference on Fundamental Approaches to Software Engineering (FASE)*, Heike Wehrheim and Jordi Cabot (Eds.). Springer International Publishing, Cham, 335–356. DOI: https://doi.org/10.1007/978-3-030-45234-6_17

Patrick Stünkel, Harald König, Adrian Rutle, and Yngve Lamo. 2021. Multi-model evolution through model repair. *J. Object Technol.* 20, 1 (2021), 1:1–25. DOI: https://doi.org/10.5381/jot.2021.20.1.a2

Patrick Stünkel, Harald König, Yngve Lamo, and Adrian Rutle. 2018. Multimodel correspondence through inter-model constraints. In *2nd International Conference on Art, Science, and Engineering of Programming (Programming)*. ACM, 9–17. DOI: https://doi.org/10.1145/3191697.3191715

The Linux Foundation. 2021. OpenAPI Initiative. Retrieved from https://www.openapis.org/.

Frank Trollmann and Sahin Albayrak. 2015. Extending model to model transformation results from triple graph grammars to multiple models. In *8th International Conference on Theory and Practice of Model Transformations (ICMT)*. Springer International Publishing, 214–229. DOI: https://doi.org/10.1007/978-3-319-21155-8_16

Frank Trollmann and Sahin Albayrak. 2016. Extending model synchronization results from triple graph grammars to multiple models. In *9th International Conference on Theory and Practice of Model Transformations (ICMT)*. Springer International Publishing, 91–106. DOI: https://doi.org/10.1007/978-3-319-42064-6_7

Bert Vanhooff, Dhouha Ayed, Stefan Van Baelen, Wouter Joosen, and Yolande Berbers. 2007. UniTI: A unified transformation infrastructure. In *10th International Conference on Model Driven Engineering Languages and Systems (MODELS)*. Springer, Berlin, 31–45. DOI: https://doi.org/10.1007/978-3-540-75209-7_3

Dániel Varró, Szilvia Varró-Gyapay, Hartmut Ehrig, Ulrike Prange, and Gabriele Taentzer. 2006. Termination analysis of model transformations by Petri nets. In *3rd International Conference on Graph Transformations (ICGT'06) (Lecture Notes in Computer Science)*, Vol. 4178. Springer, Berlin, 260–274. DOI: https://doi.org/10.1007/11841883_19

Jens von Pilgrim, Bert Vanhooff, Immo Schulz-Gerlach, and Yolande Berbers. 2008. Constructing and visualizing transformation chains. In *4th European Conference on Model Driven Architecture – Foundations and Applications (ECMDA-FA)*. Springer, Berlin, 17–32. DOI: https://doi.org/10.1007/978-3-540-69100-6_2

Dennis Wagelaar. 2008. Composition techniques for rule-based model transformation languages. In *1st International Conference on Theory and Practice of Model Transformations (ICMT)*. Springer, Berlin, 152–167. DOI: https://doi.org/10.1007/978-3-540-69927-9_11

Dennis Wagelaar, Massimo Tisi, Jordi Cabot, and Frédéric Jouault. 2011. Towards a general composition semantics for rule-based model transformation. In *14th International Conference on Model Driven Engineering Languages and Systems (MODELS)*. Springer, Berlin, 623–637. DOI: https://doi.org/10.1007/978-3-642-24485-8_46

Dennis Wagelaar, Ragnhild Van Der Straeten, and Dirk Deridder. 2010. Module superimposition: A composition technique for rule-based model transformation languages. *Softw. Syst. Model.* 9, 3 (June 2010), 285–309. DOI: https://doi.org/10.1007/s10270-009-0134-3

Nils Weidmann, Lars Fritsche, and Anthony Anjorin. 2020. A search-based and fault-tolerant approach to concurrent model synchronisation. In *13th ACM SIGPLAN International Conference on Software Language Engineering (SLE)*. ACM, New York, NY, 56–71. DOI: https://doi.org/10.1145/3426425.3426932

Nils Weidmann and Stefan Sauer. 2020. Applying bidirectional transformations in industrial contexts: Challenges and solutions. In *22nd Workshop Software-Reengineering & Evolution (WSRE'20)*. Gesellschaft für Informatik e.V. (GI). Retrieved from https://fg-sre.gi.de/fileadmin/FG/SRE/Forschung.

Yingfei Xiong, Hui Song, Zhenjiang Hu, and Masato Takeichi. 2013. Synchronizing concurrent model updates based on bidirectional transformation. *Softw. Syst. Model.* 12, 1 (2013), 89–104. DOI: https://doi.org/10.1007/s10270-010-0187-3

Received 31 October 2021; revised 22 April 2022; accepted 30 May 2022