Modular and Incremental Global Model Management with Extended Generalized Discrimination Networks

Matthias Barkowsky , Holger Giese

Technische Berichte Nr. 154

des Hasso-Plattner-Instituts für
Digital Engineering an der Universität Potsdam

Universität Potsdam

HPI Hasso Plattner Institut
Digital Engineering · Universität Potsdam

Technische Berichte des Hasso-Plattner-Instituts für
Digital Engineering an der Universität Potsdam

Technische Berichte des Hasso-Plattner-Instituts für
Digital Engineering an der Universität Potsdam | 154

Matthias Barkowsky | Holger Giese

Modular and Incremental Global Model Management with Extended Generalized Discrimination Networks

Universitätsverlag Potsdam

Bibliographic information published by the Deutsche Nationalbibliothek
The Deutsche Nationalbibliothek lists this publication in the Deutsche Nationalbibliografie; detailed bibliographic data are available on the Internet via `http://dnb.dnb.de/`.

Universitätsverlag Potsdam 2023
`http://verlag.ub.uni-potsdam.de/`

Am Neuen Palais 10, 14469 Potsdam
Phone: +49 (0)331 977 2533 / Fax: 2292
Email: verlag@uni-potsdam.de

The series **Technische Berichte des Hasso-Plattner-Instituts für Digital Engineering an der Universität Potsdam** is edited by the professors of the Hasso Plattner Institute for Digital Engineering at the University of Potsdam.

ISSN (print) 1613-5652
ISSN (online) 2191-1665

Layout: Tobias Pape
Print: docupoint GmbH Magdeburg

ISBN 978-3-86956-555-2

Also published online on the publication server of the University of Potsdam:
`https://doi.org/10.25932/publishup-57396`
`https://nbn-resolving.org/urn:nbn:de:kobv:517-opus4-573965`

Complex projects developed under the model-driven engineering paradigm nowadays often involve several interrelated models, which are automatically processed via a multitude of model operations. Modular and incremental construction and execution of such networks of models and model operations are required to accommodate efficient development with potentially large-scale models. The underlying problem is also called Global Model Management.

In this report, we propose an approach to modular and incremental Global Model Management via an extension to the existing technique of Generalized Discrimination Networks (GDNs). In addition to further generalizing the notion of query operations employed in GDNs, we adapt the previously query-only mechanism to operations with side effects to integrate model transformation and model synchronization. We provide incremental algorithms for the execution of the resulting extended Generalized Discrimination Networks (eGDNs), as well as a prototypical implementation for a number of example eGDN operations.

Based on this prototypical implementation, we experiment with an application scenario from the software development domain to empirically evaluate our approach with respect to scalability and conceptually demonstrate its applicability in a typical scenario. Initial results confirm that the presented approach can indeed be employed to realize efficient Global Model Management in the considered scenario.

Contents

1 Introduction

Complex projects developed under the model-driven engineering paradigm nowadays often involve several interrelated models, which are inspected, analyzed, transformed, and synchronized via a multitude of model operations **[66]**[1]. An effective and efficient management of the resulting sophisticated networks of model operations is both a crucial prerequisite to successful development projects and a challenging research problem, known as Global Model Management [9].

On the one hand, modular and incremental construction of model operation networks is required in the context of project landscapes that evolve to accommodate dynamic development processes and changing requirements. On the other hand, in order to scale to today's potentially large models and allow development in teams, modular and incremental execution of these networks is required, as full re-execution of the entire network in reaction to changes may result in unacceptable execution times and loss of information **[39]**.

In this context, model queries, due to being explicitly and implicitly required by model properties and model consistency checks respectively model transformations and model synchronizations, play a central role. Solutions thus have to offer dedicated support for handling potentially complex model queries and facilitate their modular composition and reuse.

Furthermore, model operations with side-effects, such as model transformation and synchronization, and their interaction with other model operations pose a unique challenge regarding the overall goal of guaranteeing the consistency of a system description that may be distributed over multiple models.

In this report, we propose an approach to Global Model Management that specifically aims to provide both the required modularity and incrementality. Our solution is based on an extended notion of Generalized Discrimination Networks [43], a mechanism that has previously been implemented in the context of model driven engineering **[6]** to allow a modular and incremental specification and execution of model queries in the form of nested graph conditions [42].

Therefore, we introduce a more general formalization called *extended Generalized Discrimination Networks (eGDNs)*, which (i) supports a more flexible notion of model queries, affording increased expressiveness and (ii) allows the integration of model operations with side effects into the unifying framework. In addition, we provide algorithms for the incremental execution of eGDNs.

Furthermore, we integrate a number of typical model operations into a prototypical implementation of the approach and use this implementation to perform an

---

[1] Note that references in bold refer to our own publications.

initial evaluation of our technique's scalability using an application scenario from the software development domain. This empirical evaluation is complemented by a conceptual evaluation regarding the applicability of eGDNs in a typical scenario.

The remainder of the report is structured as follows: We briefly reiterate the basic concepts of models in the form of typed graphs and discrimination networks in Chapter 2. After introducing the required concepts, we discuss requirements of a solution for global model management and related work in Chapter 3, providing further motivation for the design of a new solution. Our contribution in the form of extended Generalized Discrimination Networks is presented in Chapters 4, 5, and 6. Therefore, Chapter 4 provides a definition of eGDNs along with a graphical notation. Chapter 5 describes the incremental execution of eGDNs. Chapter 6 then lists a number of examples for eGDN operations that are part of our prototypical implementation. This prototypical implementation is used to perform an initial empirical evaluation of the presented concepts, which is presented in Chapter 7 along with a conceptual evaluation of the applicability of eGDNs to an example use case. Finally, Chapter 8 concludes the report and gives an overview of possible directions for future work.

2 Preliminaries

In this chapter, we reiterate the basic notions of models in the form of typed graphs and discrimination networks.

2.1 Graphs and Models

A graph $G = (V^G, E^G, s^G, t^G)$ consists of a set of vertices V^G, a set of edges E^G, and two functions $s^G, t^G : E^G \rightarrow V^G$ assigning each edge its source respectively target vertex [24]. A graph morphism $m : G \rightarrow H$ between graphs G and H is a pair of functions $m^V : V^G \rightarrow V^H, m^E : E^G \rightarrow E^H$ such that $s^H \circ m^E = m^V \circ s^G$ and $t^H \circ m^E = m^V \circ t^G$.

A graph G can be typed over a type graph TG via a morphism $type^G : G \rightarrow TG$ that assigns elements from G types defined in TG. This yields a typed graph $G^T = (G, type^G)$. A typed graph morphism $m^T : G^T \rightarrow H^T$ between two typed graphs $G^T = (G, type^G)$ and $H^T = (H, type^H)$ typed over the same type graph TG is given by a graph morphism $m : G \rightarrow H$ with $type^G = type^H \circ m^T$.

In the context of this report, a model is then characterized by a typed graph, where the type graph effectively acts as a metamodel. Importantly, attributes for model elements can be realized in the framework of typed graphs by simply modeling attribute values as dedicated nodes, which leads to the notion of typed attributed graphs [45]. A modeling language ML is defined by a graph TG and denotes the set of all possible graphs typed over TG.

Figure 2.1 shows an example model from the software development domain in the form of a typed graph G, and the associated metamodel in the form of the type graph TG, with the typing morphism given by node labels in case of nodes and implicitly in case of edges. The example model represents the abstract syntax graph (ASG) of a program written in an object-oriented programming language. Nodes in the model represent packages, types, and methods. Edges represent containment relationships between the different concepts, with methods contained in types and types contained in packages, and return type relationships between methods and types.

2.2 Discrimination Networks

A discrimination network is a graph of nodes representing computation units and edges representing dependencies between these units. Discrimination networks

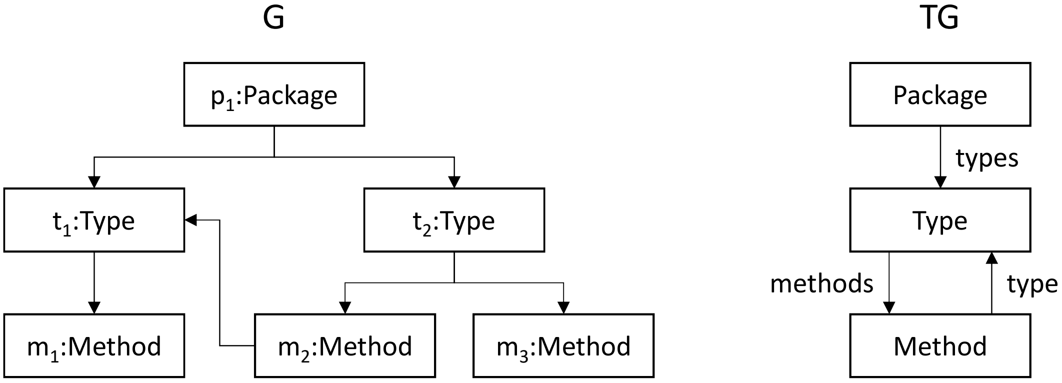

Figure 2.1: Example model and metamodel in the form of typed graph and type graph from the software development domain

are a popular solution for the incremental execution of model queries such as the computation of model properties or the checking of model consistency conditions. Therefore, the model query is decomposed into subqueries, which form the discrimination network's nodes.

The execution of a subquery can make use of the results computed for another subquery, which is indicated by a dependency relation between the two subqueries. The execution of a final discrimination network node yields the overall query result. By storing the results of discrimination network nodes beyond the execution of a query, incremental execution that reuses previously computed results in subsequent executions is enabled.

Since discrimination networks so far are primarily employed for model querying, current approaches offer only limited or no support for the integration of model operations with side-effects and thus constitute at best a partial solution for global model management. However, due to their inherent support for modularity and incrementality, they offer a promising starting point.

There exist different realizations of the concept of discrimination networks in the context of model driven engineering, two of which will be briefly presented in the following subsections.

2.2.1 RETE nets

RETE nets were initially introduced by Forgy [34] and are characterized by the fact that nodes are only allowed to have dependencies to at most two other nodes. Some examples of RETE nodes are:

- *input nodes*, which correspond to primitive model queries that extract individual elements, that is, nodes or edges, from a model, and consequently have no dependencies
- *filter nodes*, which filter the results of some other subquery by a condition and consequently have one dependency

- *join nodes*, which combine the results of two other subqueries into results for a more complex subquery and consequently have two dependencies

While the listed node types form the core of incremental model querying solutions such as the well-established VIATRA [73], RETE nets are a flexible mechanism that allows a multitude of other query-related node types. This is illustrated by VIATRA's support for various advanced constructs for specifying model queries, including negative patterns and certain aggregation operations.

In RETE implementations, results computed by a RETE net's nodes are usually stored in memory in so-called indexers, which act as implicit interfaces between computation nodes. These indexers can also be made explicit by modeling them as part of the RETE net via a different kind of RETE node that is not associated with any computational functionality, but only serves as a storage for other nodes' results.

2.2.2 Generalized Discrimination Networks

Generalized Discrimination Networks (GDNs) are a less restrictive form of discrimination networks than RETE nets and were developed by Hanson et al. [43]. Essentially, GDNs drop the limit on the number of a node's dependencies of RETE nets and thereby allow for more control over which intermediate query results are to be stored in memory.

A realization in the context of model querying was presented in **[6]**. It implements GDN nodes as model transformation rules that create marking elements for subquery results directly as part of the queried model. Dependencies between nodes are realized by considering marking elements created by the required node in the transformation rule associated with the dependent node. However, while the approach in **[6]** is based on a fairly expressive notion of queries in the form of nested graph conditions, certain query-related operations such as aggregation are not supported by the underlying formalism.

3 Requirements for Global Model Management

Nowadays the development of complex systems with models requires *Global Model Management* (*GMM*) [8, 31] to ensure that the models of different subsystems, of different views, and of different domains are properly combined, even though the models might reside at different levels of abstraction. Indeed, due to the heterogeneity and complexity of systems such as Cyber-Physical Systems (CPS), it is no longer feasible to represent the system as a Single Underying Model (SUM). This is because numerous languages and tools are already employed independently by domain experts collaborating to build the system. Redeveloping these tools and thus requiring industry to change its practices is not conceivable given the required development efforts, but also the strong resistance to change development processes. This is especially relevant in the case of safety-critical systems that must undergo complex certification processes. Therefore, many models must be used to represent the system and adequate GMM is required to ensure that the development activities that operate on the models are properly coordinated such that the models lead to a proper system as a whole, where the different elements and aspects covered by the different models are correctly integrated and are consistent with each other.

A classification of model integration problems and fundamental integration techniques has been introduced in **[38]**. It highlights the techniques of decomposition and enrichment, which characterize two orthogonal dimensions of development where the system is decomposed into subsystems and domains (*horizontal* dimension) and into a set of models with increasing level of details (*vertical* dimension). This requires coordinating all activities operating on the models across these dimensions to ensure their consistency.

The development activities for nowadays complex systems are spread across multiple domains and teams, where each team is using its own set of modeling languages thus requiring proper *integration* of these languages. Indeed, it has been shown that using a single language to cover all domains would lead to very large monolithic languages not easily customizable for the development environments and tools needed by development organizations. These considerations lead to *Multi-Paradigm Modeling* (*MPM*) [72], which advocates the integration of reusable *modular* modeling languages instead of large monolithic languages. Hence, GMM must support integrating with appropriate modularity not only models but also their *modeling languages* (hereafter *modeling language integration*), in addition to coordinating all activities operating on the models and specified as *model operations / transformations*. The execution of these model operations has to be *scalable* for being able to handle large models. This requires *incrementality*, where only the operations impacted by a model change are re-executed, thus avoiding the effort to recompute entire models as in the case of incremental code compilers.

GMM is also known as *modeling-in-the-large*, which consists of establishing global relationships (e.g. model operations that generated one model from other models) between macroscopic entities (models and metamodels) while ignoring the internal details of these entities [8]. *Megamodeling* [10, 31] has been introduced for the purpose of describing these macroscopic entities and their relations.

Consequently, for modular and incremental global model management solutions for the modular and incremental construction and execution of I) models and modeling languages integration, II) model operations, and III) megamodels are required. We will outline in the following that nowadays only preliminary approaches exist that provide ad hoc solutions for fragments of the sketched problem and that a solid understanding of the underlying needs and challenges is currently lacking. In particular, the current approaches do at most offer some modularity and/or incrementality for a single aspect as modeling languages integration or model operations. However, support for handling complex modeling landscapes as a whole in a modular and incremental fashion as required for the large-scale problems that exist in practice is not offered so far.

In the following, we will discuss the needs in more detail and review how far existing solutions that address the construction and execution of 1) models and modeling languages integration, 2) model operations, and 3) megamodels. The way the existing approaches perform along these dimensions is depicted in Table 3.1, where an empty cell identifies a need that is not addressed, a ~ denotes partial fulfilment of the need and a + indicates that the need is addressed sufficiently.[1] This evaluation is discussed in further details in the following sections.

3.1 Models and Modeling Languages Integration: Construction and Execution

3.1.1 Construction

The construction of models and modeling languages integration is addressed in the current approaches in three main ways via (1) linking of models and model elements, (2) model interfaces and (3) metamodel composition.

(1) Links:

All approaches make use of some kind of *trace links* between models and their model elements to integrate models. In this report, we adopt the definitions of traceability proposed by the Center of Excellence for Software Traceability (CoEST) [20].

---

[1] For convenience, we use the name of the tool or project to identify an approach when it exists, otherwise the name of the first author of the publication describing the approach is used.

Approach	**Modeling Languages Integration**					**Model Operations**				**Megamodels**			
	Const.			*Exec.*		*Const.*		*Exec.*		*Const.*		*Exec.*	
	Links	*Int.*	*MMI*	*Batch*	*Inc.*	*Flow*	*Ctx.*	*Batch*	*Inc.*	*Mon.*	*Mod.*	*Batch*	*Inc.*
Modeling Languages Integration													
Blanc et al. [11]				+	+								
EMF IncQuery [25, 71]				+	+								
Egyed et al. [23, 41]				+	+								
Cabot et al. [19]				+	+								
ACOL [54]		~		+									
SmartEMF [48, 55]	+	+											
Composite EMF Models [21, 50]	+	+											
EMF Views [18, 27]	+		~	+									
Kompren [12] / Kompose [33, 52]			~										
Reuseware ModelSoc [49]		~	~										
Ratiu et al. [61]	+			~	~								
König et al. [53]	+			+									
Model Operations													
Wires* [62]						+		+					
ATL Flow [4]						+		+					
Epsilon [28, 60]	+					+		+					
Gaspard2 [29]			~			+	~	+	~				
Debreceni et al. [22]						+	~	+	+				
MoTCoF [65]						+	+	+	~				
MoTE [37][57]	+							+	+				
Integration Languages and Others													
CyPhy [68]	+					+		+					
FUSED [15, 35]	+	~				+		+					
CONSYSTENT [46, 47]	+					+		+					
Megamodels													
AM3 [1, 74]	+					+		+		+		~	
FTG+PM [5, 56]						+		+		+		+	
MegaL Exp. [32]	+					+				+			
GMM* [13]	+		~			+		+	~	+		+	~
Seibel et al. [64, 67][7]	+									+		+	~
Stevens [69, 70]						+		+	+	+		+	+
Gleitze et al. [40]						+		+	+	+		+	+
Vitruvius [51]	+			+	+	+		+	+	+		+	+
eGDNs	+	+	+	+	+	+	+	+	+	+	+	+	+

Table 3.1: Comparison of existing and planned global model management approaches

A *trace link* is *"...a specified association between a pair of artifacts, one comprising the source artifact and one comprising the target artifact..."*. Following the CoEST again, trace links are specialized into traces between the *vertical* and *horizontal* dimensions. Hence, a *vertical* trace *"...links artifacts at different levels of abstraction so as to accommodate lifecycle-wide or end-to-end traceability, such as from requirements to code..."*. A horizontal trace links *"...artifacts at the same level of abstraction, such as: (i) traces between all the requirements created by 'Mary', (ii) traces between requirements that are concerned with the performance of the system, or (iii) traces between versions of a particular requirement at different moments in time"*.

There is a plethora of approaches (e.g., [2, 15, 28, 32, 46, 55, 68] **[57]**) making use of trace links to integrate models. The Atlas Model Weaving (AMW) language [2] provided one of the first approaches for capturing hierarchical traceability links between models and model elements. The purpose was to support activities such as automated navigation between elements of the linked models. In this approach, a generic core traceability language is made available and optionally extended to provide semantics specific to the metamodels of the models to be linked. Similarly, the Epsilon framework [28] provides a tool (ModeLink) to establish correspondences between models. MegaL Explorer [32] supports relating heterogeneous software development artifacts which do not necessary have to be models or model elements using predefined relation types. SmartEMF [55] is another tool for linking models based on annotations of Ecore metamodels to specify simple relations between model elements through correspondence rules for attribute values. Complex relations are specified with ontologies relating the concepts of the linked languages. The whole set of combined models is converted into Prolog facts to support various activities such as navigation, consistency and user guidance when editing models. The CONSYSTENT tool and approach [46] make use of a similar idea. However, graph structures and pattern matching are used to represent the combined models in a common formalism and to identify and manage inconsistencies instead of Prolog facts as in the case of SmartEMF.

There are also a number of approaches such as [68] and [15] that build on establishing links between models through the use of integration languages developed for a specific set of integrated modeling languages, where the integration language embeds constructs specific to the linked languages. This is also the case for model weaving languages extending the core AMW language. However, AMW has the advantage of capturing the linking domain with a core common language. Other means for linking and integrating models are Triple Graph Grammars (TGG) such as the Model Transformation Engine (**MoTE**) tool **[57]**, which similarly requires the specification of some sort of integration language (correspondence metamodel) specific to the integrated languages. However, an important asset of this approach is that it automatically establishes and manages the traceability links and maintains the consistency of the linked models (model synchronization) in a *scalable*, *incremental* manner. Finally, in **[7, 64, 67]**, an approach is presented to automatically create and maintain traceability links between models in a scalable manner. While the approach focuses on traceability management rather than model integration, compared to integration languages, it relies on link types defined at the model level (and not at the metamodel / language level), thus avoiding the need to update the integration language every time a new language must be integrated. Recently, the concept of reactive links has been presented [61], which allows incremental propagation of attribute value changes between models of different languages. However, incremental execution is only offered for a limited notion of consistency.

The comparison of these approaches shows that apart from our earlier approach **[7, 64, 67]**, all approaches suffer from being dependent on the set of integrated languages, thus requiring to better support modularity. Furthermore, only our own work **[7, 57, 64, 67]** supports automated management of traceability links.

(2) Interfaces:
In addition to links, a few more sophisticated approaches (e.g., [48, 49, 50, 54]) introduce a concept of *model interface* (*int.* column in Table 3.1) for specifying how models can be linked. In [54], the Analysis Constraints Optimization Language (ACOL) is proposed, which has been designed to be pluggable to an Architecture Description Language (ADL). A concept of *interface* specific to ACOL is included so that constraints can refer to these interfaces to relate to the model elements expected from the ADL. SmartEMF [48] proposes a more generic concept of model interface to track dependencies between models and metamodels and provide automated compatibility checks. Composite EMF Models [21, 50] introduces *export* and *import* interfaces to specify which model elements of a main model (*body*) should be exposed to other models (i.e. are part of the public API), and which elements of a body model are to be required from an export interface. In [49], an approach for the composition of grammars with explicit variation points (hooks) constituting an implicit *invasive* composition interface is presented.

However, while these approaches provide interesting preliminary ideas, they need to be enriched to cover a larger number of non intrusive model integration use cases such as for example, specifying modification policies of the linked model elements required to ensure the models can be kept consistent. They also lack integration into GMM.

(3) Metamodel Integration:
Some approaches (e.g., [13, 27, 29, 52]) consider the construction of view metamodels in terms of other metamodels or language fragments (*MMI* column in Table 3.1). In [29], an approach implemented in the Gaspard2 tool is presented where metamodels are artificially extended for the purpose of combining independent model transformations resulting in an extended transformation for the extended metamodels. In [12], a language and tool (Kompren) are proposed to specify and generate slices of metamodels via the selection of classes and properties of an input metamodel. A reduced metamodel is then produced, which must be completely regenerated when the input metamodel is changed. Such is the case for the Kompose approach [52], which on the contrary to Kompren proposes to create *compound metamodels,* where a set of visible model elements from each combined metamodels is selected and optionally related. EMF Views [18, 27] provides similar approach however without the need to duplicate the metamodel elements as opposed to Kompose and Kompren. Indeed, EMF Views allows the specification of *virtual* metamodels that only refer to existing metamodel elements instead of duplicating them. The same principle applies for the given models of the virtual metamodels, which only refer to elements of the existing integrated models instead of duplicating them. The defined virtual view metamodels are usable transparently by tools. Furthermore, the same models can be simultaneously used by both legacy tools and new tools making use of the virtual metamodels, thanks to the non-intrusiveness of the approach. Finally, the Global Model Management lan-

guage (**GMM***)[2] [13] provides means to specify and interpret reusable language subsets as sets of constraints combined to form subsetted metamodels. Like for EMF Views, these reduced metamodels can to some extent be used transparently by tools. Aspect-oriented metamodel composition is another well-known technique for metamodel composition. However it requires metamodels to be expressed in a specific aspect-related format, which does not meet our non-intrusiveness requirement.

While each of these approaches provides interesting support for modular modeling languages integration, their unification into a common formalism, the use of an explicit notion of a model interface and their integration into GMM is lacking, except for subsetted metamodels already integrated within our **GMM*** language. Among these approaches, we note that EMF Views provides an adequate starting point for this work, due to its *non-intrusiveness* property essential for reusing legacy models and tools. However, in its current implementation, only changes of attributes of virtual compound models are propagated to the underlying real models [18]. Other changes propagation as well as metamodel constraints composition remain to be addressed. The integration of an explicit metamodel interface construct for governing how metamodels can be composed, as well as the ability to solve attribute and operation conflicts of merged classes inspired from the concept of *Traits / Mixins* developed for object oriented programming are required future works for this approach.

Execution of integrated models concerns the evaluation of the well-formedness constraints of each combined model alone, but also of the combined models as a whole. To our knowledge, no approach addresses the incremental checking of well-formedness conditions across the different language fragments of compound models. However, some approaches on incremental constraints evaluation exist. In [11], changes on models are expressed as sequences of atomic model operations to determine which constraint is impacted by the changes, so that only these constraints need to be re-evaluated. In [25, 71], a graph-based query language (EMF-IncQuery) relying on incremental pattern matching for improved performance is also proposed. In [23], an approach is presented for incremental evaluation of constraints based on a scope of model elements referenced by the query and determined during the first query evaluation. This scope is stored into cache and used to determine which queries need to be re-evaluated according for some model changes. In [41], this approach is extended for the case where the constraints themselves may change besides the constrained models. Finally in [19], an incremental OCL checker is presented where a simpler OCL expression and reduced context elements set are computed from an OCL constraint and a given structural change event. Evaluating this simpler constraint for the reduced context is sufficient to assert the validity of the initial constraint and requires significantly less computation resources.

---

[2] We use * to distinguish this existing language and tool from the generic Global Model Management (GMM) acronym.

In [53], König et al. introduce a technique for the checking of consistency constraints over linked models, which avoids the merging of these models into a single underlying model to achieve better scalability. However, while formally defined and proven to be correct, the approach in [53] does not consider incremental consistency checking.

We identified the following requirements as main needs concerning modularity and incrementality of modeling languages integration:

> **R 1.1** modeling languages integration via integration links and combination of well-formedness conditions with consistency
>
> **R 1.2** interfaces for embedding of modeling languages

Note that concerning Table 3.1 the requirements cover here Links and Interfaces which jointly emulate the less modular direct meta model integration and that the employed well-formedness conditions and consistency conditions will be covered when we consider model operations in the next section. Consequently, as visible in Table 3.1, there yet does not exists any approach that provides a combination of all these requirements we target.

3.2 Model Operations: Construction and Execution

The construction of model operations is addressed in two ways in the literature. Most approaches combine model operations as *model transformations chains* ((1) *Flow Composition*), where each chained transformation operates at the granularity of complete models. In order to support reuse and scalability for complex modeling languages, which are defined by composing them from simpler modeling languages, a few approaches have considered specifying model transformations as white boxes. Composed of explicit fine grained operations processing model elements for a given context, these operations are reusable across several model transformations ((2) *Context Composition*).

(1) Flow Composition Approaches:

FUSED (Formal United System Engineering Development) [15] is an integration language to specify complex relationships between models of different languages. It supports model transformation chains, but only implicitly via execution of tools, without explicit representation of the involved transformations and processed data. On the contrary, there is a plethora of approaches allowing the explicit specification and construction of model transformation chains implementing a data flow paradigm. Such is the case of the AtlanMod Megamodel Management (AM3) tool [1], for which the Atlas Transformation Language (ATL) [3] is used to specify the model transformations. Besides, a type system has been developed [74], which enables type checking and inference on artifacts related via model transformations. Another similar but less advanced tool is the Epsilon Framework [28], which provides model transformation chaining via ANT tasks. Wires [62] and ATL Flow [4] are tools providing graphical languages for the orchestration of ATL model transformations. The Formalism Transformation Graph + Process Model (FTG+PM) formalism [56] implemented in the AToMPM (A Tool for Multi-Paradigm Modeling) tool [5] provides similar functionality. However, it has the advantage of also specifying the complete modeling process in addition to the involved model transformations. This is achieved via activity diagrams coupled with model transformation specifications executed automatically to support the development process. Finally, **GMM*** [13] also supports model transformation chaining, but through the specification of relations between models of specific metamodels that can be chained. One advantage of this approach is that automated incremental (re-)execution of the specified relations between models is provided in response to received model change events. Incrementality of the execution of the transformations is also made possible by the integration of the **MoTE [57]** incremental model transformation tool into **GMM***.

However, while chaining model transformations offers some degree of modularity of model transformation specifications, apart from **GMM***, most approaches suffer from scalability issues for large models, since the used transformation tools do not support incremental execution. In addition, the case where a generated model is modified by hand to add information not expressible with the language of the original model(s) cannot easily be handled by these approaches, since regenerating the model modified by hand will destroy the user-specific information. This need is better supported by context composition approaches.

(2) Context Composition Approaches:

A few approaches allow context composition of model operations (column *Ctx.* in Table 3.1). In [29] as mentioned above, an approach is described to combine independent model transformations resulting in extended transformations for corresponding extended metamodels. In [22], an approach is described for specifying the construction of view models using contextual composition of model operations (derivation rules) encoded as annotations of queries of the EMF IncQuery [25] language. Traceability links between view and source model elements are automatically established and maintained. The use of EMF IncQuery natively provides incremental execution of the derivation rules to synchronize the view model with the source model. Some views may be derived from other views thus allowing flow composition as chains of view models. This approach achieves results similar to TGGs supporting incrementality, however with the drawback of being unidirectional. Similarly, but with bi-directionality the **MoTCoF** language **[65]** allows for both flow and fine grained context composition of model transformations. An advantage over [29] however is that model transformations are used as black boxes without the need to adapt the transformations according to the context.

As can be seen, most approaches only support flow type modularity for model operations with batch execution except for our **GMM*** language thanks to its integration of **MoTE** providing incremental execution. This will not scale and lead to information losses in case of partial model information overlap. Only a few approaches allow context modularity, which better supports incremental application where only the impacted operations can be re-applied following a change in order to avoid the cost of re-computing complete transformations. Such is the case of **MoTCoF**, which theoretically permits incremental execution, but a concrete technical solution is still lacking for it.

To address modularity and incrementality for model operations, we identified as main needs:

R 2.1 composition of model operations

R 2.2 model operations over integrated models

R 2.3 execution scheme for model operations

Note that concerning Table 3.1 the requirements cover here Flow and Context based composition and Batch as well as Incremental Execution at first for all special cases of model operations and then also for the general case. Consequently, as visible in Table 3.1, there yet does not exists any approach that fully cover the envisioned combination of all these requirements we target.

3.3 Megamodels and other Global Model Management Approaches

Two strands can be identified for GMM. A first one makes use of *(1) model integration languages*, which are defined for a specific set of integrated modeling languages and tools meaning that the integration language must be updated every time a new language or tool is used. The second strand attempts to solve this problem by making use of *(2) megamodels* providing configurable global model management.

(1) Integration Languages and other Approaches:

The CyPhy [68] used in the GME modeling tool and FUSED [15, 35] are examples of model integration languages. But as mentioned above, these languages must be adapted as soon as a different set of integrated languages and tools must be used, thus requiring highly skilled developers. Integration languages are therefore not practical.

Open Services for Lifecycle Collaboration (OSLC) [59] provides standards for tool integration through the Web. Many specifications are available for *change management*, *resource previews*, *linked data*, etc. It builds on the W3C *linked data* standard, which aims at providing best practices for publishing structured data on the Web based on the W3C Resource Description Framework (RDF). RDF is a model for data interchange on the Web where data is represented as graphs. However, OSLC is more services (and tools) oriented and inherits the problems of *linked data*, which is specific to the Web and therefore does not separate the concerns of data representation and persistence as opposed to Model-Driven Engineering (MDE) where an abstract syntax is used independently of the way the data is stored.

Another approach making use of these standards is [46] and is implemented in a tool named CONSYSTENT, used to identify and resolve inconsistencies across viewpoints due to information overlapping. The information of all models involved during development is captured in a common RDF graph. The approach relies on a human[3] to specify patterns representing semantic equivalence links (semantic connections) across the graph models. Inconsistency patterns based on these semantic connections are continuously checked over the RDF model for potential matches identifying inconsistencies. Means to automatically resolve inconsistencies are under development. However, this approach necessitating the conversion of all models as a RDF graph is not incremental and will not scale for large models.

(2) Megamodels:

In this second strand, megamodels serve to capture and manage MDE resources such as modeling languages, model transformations, model correspondences, and tools used in modeling environments. There are several megamodeling approaches as already mentioned. AM3 [1] is one of the first ones where a megamodel is basically a registry for MDE resources. Model transformations are specified with ATL [3] and model correspondences with the Atlas Model Weaving (AMW) language [2]. Similarly, FTG+PM [56] as mentioned above is also a megamodeling language as well as MegaL Explorer [32] allowing to model the artifacts used in software development environments and their relations from a linguistic point of view. The involved software languages and related technologies and technological spaces can be captured with linguistic relationships between them such as membership, subset, conformance, input, dependency, definition, etc. Operations between entities can also be captured. The artifacts do not need to be represented as models, but each entity of the megamodel can be linked to a Web resource that can be browsed and examined. However, the language seems to be used mostly for visualization providing a better understanding of the developments artifacts but cannot be executed to perform model management. The aforementioned **GMM*** infrastructure [13] consists of a megamodeling language inspired from **[44]**. Metamodels can be declared, as well as relations between models of these metamodels. In particular, synchronization relations can relate models of two different metamodels making use of the **MoTE** TGG engine **[57]** to transform or synchronize the models. As mentioned earlier, chains of model transformations can be specified and executed incrementally in response to model change events and *subsets* of modeling languages can be declared. **GMM*** is experimented within the Kaolin tool [14] making use of complex and rich industrial languages such as AADL and VHDL thus challenging GMM for realistic specifications.

---

[3] An automated method making use of Bayesian Belief Networks is also under study [47].

A new approach to modeling in the large with bidirectional model transformation has been proposed by Stevens [70]. The work in [70] presents a formalized notion of a megamodel in the form of a hypergraph, where models are represented as nodes that can be connected via hyperedges representing bidirectional transformations. Incremental execution is generally supported by the formalism, however, a concrete algorithm is only presented for megamodels with a restricted structure for which a certain notion of correctness can be guaranteed. The author extends her work in [69] by connecting her previous work to research in the domain of build systems and introducing a so-called orientation model to steer megamodel execution, relaxing the restrictions on the megamodel's structure while maintaining a formal guarantee of correctness. However, the construction of an orientation model is a manual and potentially challenging process for complex networks of model operations. Furthermore, the work in [69, 70] abstracts from the technical realization of model operations and hence does not explicitly consider how operations such as the computation of model properties may be composed in a modular manner.

In [40], Gleitze et al. propose an incremental execution strategy for networks of model transformations, specifically aiming for a solution that provides explanations of cases where the strategy failed to produce a consistent result. While their strategy is applicable to networks with arbitrary structure, only bidirectional transformations between pairs of models are considered, limiting the notion of supported model operations.

Recently, significant progress has also been made in the field of model views [16], which studies how consistent view models can be derived from a system description consisting of multiple interrelated models and therefore also relates to Global Model Management. The most comprehensive and advanced model view technique is probably the Vitruvius approach [51], which relies on a so-called virtual single underlying model (V-SUM) for the description of the overall system under development. The V-SUM is used to integrate the individual models describing system parts and derive new view models via consistency relations. Therefore, Vitruvius employs a dedicated incremental algorithm for executing complex networks of consistency preservation operations. However, the notion of consistency in [51] is limited to relations between pairs of tuples of model elements and hence does not support certain model operations such as computation of model properties using aggregations. Furthermore, intra-model well-formedness is deliberately not covered and reuse at the mega-model level is not considered.

However, most of these megamodeling approaches only cover to a certain degree the core ingredients of specifying MDE resources by means of metamodels and model operations with appropriate modularity and incrementality. Only fragments of the problem are solved. Furthermore, all these megamodeling languages are monolithic (column *Mon.* in Table 3.1) and as a result, predefined megamodel fragments cannot be composed and reused to avoid rebuilding complete megamodel specifications from scratch for new projects. We note however that aspect-oriented metamodel composition may be used as an inspiring point and adapted to megamodeling for the specification of distributed megamodels fragments contributing cross-cutting information in an integrated megamodel. As for megamodel execution, FTG+PM, [40, 51, 70], **GMM***, and **[64, 67]** consider automated or semi-automated execution in response to model changes or modeling events from the tool's user interfaces.

The related work demonstrates that for global model management, we need a view that combines all its facets in a mega model. To address modularity and incrementiality for modamodels we can conclude that the main needs are:

R 3.1 a megamodeling language with

R 3.1.1 support for metamodels, well-formedness, model operations, integration views, and traceability links

R 3.1.2 a megamodel operation module concept

R 3.2 a robust incremental megamodel execution scheme

R 3.3 megamodel interfaces

R 3.4 an asynchronous incremental megamodel execution scheme

Note that concerning Table 3.1, the requirements cover here the modular construction as well as incremental execution. As visible in Table 3.1 there do exist three approaches that do not support modularity but provide a combination of all the other requirements we target. However, neither of them provides the required robust incremental megamodel operation execution scheme. The technique in [69, 70], while providing formal guarantees regarding correctness and termination, is limited to networks of model operations in the form of trees of synchronizations between pairs of models or requires the manual construction of an orientation model. The Vitruvius approach [51], by virtue of employing a fixpoint iteration, does not introduce any restrictions regarding the network's structure, but consequently does not guarantee termination. The execution scheme presented in [40] is applicable to networks of model synchronizations between pairs of models with arbitrary structure and also guarantees termination. However, outside of performing the actual execution on the concrete instance, it provides no means of determining whether a network will eventually terminate with a correct result.

3.4 Summary of the state of the art

This survey of the state of the art demonstrates that several approaches address the needs for modularity and incrementality raised in this report. However, none of them fulfill these needs at the three levels of model operations, modeling languages integration and megamodels that we identify as being required all at once. Moreover, for certain individual aspects of Global Model Management, solutions with adequate modularity and incrementality do not even exists yet on their own. This work specifically targets these essential needs that have not been sufficiently addressed yet.

4 Extended Generalized Discrimination Networks

In this chapter, we introduce a notion of extended Generalized Discrimination Networks (eGDNs) and explain how the new formalism can be used as a language for megamodels.

4.1 Definition of eGDNs

In order to address shortcomings of current solutions and enable the modular and incremental construction and execution of complex nets of model operations such as model properties, model consistency operations, model transformations, and model synchronization, we further generalize the idea of Generalized Discrimination Networks **[6]** to extended Generalized Discrimination Networks. Therefore, we introduce a generalized notion of GDN nodes and their interfaces. This enables the integration of model operations with side-effects and allows a more flexible definition of queries in comparison to **[6]**, which also affords increased expressiveness.

An eGDN $G = (O, S, E, s, t)$ is essentially a bipartite graph with two kinds of nodes, slot nodes and operation nodes, where O is the set of operation nodes and S is the set of slot nodes. Operation nodes can be connected to slot nodes and vice-versa via edges from the set of edges E. The source and target functions of edges are given by $s : E \to O \cup S$ respectively $t : E \to O \cup S$.

Operation nodes represent model operations or building blocks thereof, that is, suboperations. Slot nodes store information used by model operations and their suboperations in the eGDN. Edges represent dependency relationships between operation and slot nodes, with the source of an edge representing the required node and the target of the edge representing the dependent node. An operation node depending on a slot nodes indicates that the corresponding model operation uses information stored in that slot. A slot node having a dependency on an operation node means that the operation node's model operation modifies the slot's contents.

We denote the set of dependencies of a slot or operation node n in $O \cup S$ by $in(n) = \{d \in O \cup S | \exists e \in E : s(e) = d \wedge t(e) = n\}$. Similarly, we denote the set of dependent nodes of n by $out(n) = \{d \in O \cup S | \exists e \in E : s(e) = n \wedge t(e) = d\}$. G is bipartite in the sense that $\forall o \in O : in(o) \subseteq S \wedge out(o) \subseteq S$ and $\forall s \in S : in(s) \subseteq O \wedge out(s) \subseteq O$. For an operation node $o \in O$, we also refer to the set of slot nodes $in(o)$ as the *input slots* of o and to the set of slot nodes $out(o)$ as the *output slots* of o.

A slot node s is always associated with a modeling language ML or an ordered set of variables $var = \{v_1, v_2, ..., v_k\}$ and contains a model (typed graph) of ML respectively a set of variable assignments for var. A variable assignment for an ordered set of variables $var = \{v_1, v_2, ..., v_k\}$ is given by a tuple in $dom_V(v_1) \times ... \times dom_V(v_k)$, where $dom(v_i)$ denotes the domain of variable v_i, which can either be a set of nodes or edges from one or more models or a set of primitives, e.g. $\mathbb{N}$. We refer to the set of possible contained assignment sets or models of s as the slot's domain, which is given by $dom(s) = ML$ in case s is associated with a modeling language ML or by $dom(s) = \mathcal{P}(dom_V(v_1) \times ... \times dom_V(v_k))$ if s is associated with an ordered set of variables $var = \{v_1, v_2, ..., v_k\}$. Contents are then assigned to an eGDN's slots via a valuation function $val : S \rightarrow \bigcup_{s \in S} dom(s)$, such that $\forall s \in S : val(s) \in dom(s)$.

In addition to regular models, we also allow model slots to contain *linking models*. The only difference between a regular model and linking model is the fact that a linking model's set of vertices may reference vertices from other regular and linking models as edge targets, thus allowing the establishment of inter-model connections. Therefore, similarly to linking models, the metamodel of a linking model, that is, the type graph of a linking model, may refer to vertices from other type graphs as edge targets.

Regarding operation nodes, we further distinguish between *query nodes*, *transformation nodes*, and *mixed nodes*.

Query nodes extract information from models and/or other queries' results. Therefore, a query node q may have an arbitrary number of input slots and exactly one output slot. q's input slots may contain both models or sets of variable assignments, whereas q's output slot may only contain a set of variable assignments.

Transformation nodes create or modify models based on models and/or query results. Therefore, a transformation node t may have an arbitrary number of input and output slots. t's input slots may contain both models or sets of variable assignments, whereas t's output slots may only contain models.

A mixed node x constitutes a combination of query and transformation nodes and may have an arbitrary number of input and output slots, which may contain both models or sets of variable assignments.

Each operation node o with input slots $in(o) = \{s_{i_1}, ..., s_{i_k}\}$ is associated with a semantics function $\gamma_S : dom(s_{i_1}) \times ... \times dom(s_{i_k}) \rightarrow \mathcal{P}(\mathbb{F})$, where $\mathbb{F}$ denotes the set of functions $f : out(o) \rightarrow \bigcup_{s_o \in out(o)} dom(s_o)$ such that $\forall s_o \in out(o) : f(s_o) \in dom(s_o)$. Essentially, the semantics function of an operation node describes a consistency relationship between the operation's input and output slots.

To indicate that the contents of the slots adjacent to o are consistent with o's semantics function for a valuation function val, we write $o.valid(val)$. Formally, $o.valid(val) \leftrightarrow \exists f \in \gamma_S(val(s_{i_1}), ..., val(s_{i_k})) : \forall s_o \in out(o) : f(s_o) = val(s_o)$.

A valuation function val for an eGDN $G = (O, S, E, s, t)$ is consistent with G as a whole if it holds that $\forall o \in O : o.valid(val)$.

4.2 eGDNs as Megamodels

Since an eGDN encodes a network of model operations connecting a set of potentially integrated models, it represents a megamodel. The definition of eGDNs thus constitutes a language for megamodels.

Importantly, eGDNs allow the composition of model operations from nodes that realize suboperations. In addition, they also allow hierarchical composition: An eGDN (and therefore also a basic GDN or RETE net) can be interpreted as an eGDN operation node. The input and output slots are given by the input respectively output slots of its nodes that are connected to another operation node of the parent eGDN. Any slots of the child eGDN without a connection to another node of the parent eGDN can act as internal slots of the child and do not have to be exposed to the parent. However, some such potential internal slots may also be considered input or output slots if their contents are relevant to human users. The semantics function of an operation node representing a sub-eGDN is then implicitly defined by the semantics functions of that eGDN's own operation nodes.

In addition to (hierarchical) composability, eGDNs support modularity in the sense that the semantics of an operation node regarding its output slots directly depend only on the contents of its immediate input slots. Thereby, integrating additional operation nodes (along with additional slot nodes) into an eGDN only requires appropriate wiring with the node's input and output slots, but is completely independent of any other operation nodes. Effectively, slots thus act as interfaces between model operations.

eGDNs can also enable modularity at the model level by using the results of query nodes, potentially along with transformation nodes for propagating changes from the query results back to the base model, as model interfaces or views. For instance, simple projection queries in combination with access restrictions can be employed to implement different visibilities for different roles in a development process. Alternatively, dedicated view models in conjunction with bidirectional model synchronization operations can similarly serve to implement editable model views in the context of eGDNs.

Figure 4.1 shows our graphical notation for the visualization of eGDNs. Slot nodes are depicted as rectangles and labelled "A" in the top right corner in the case of assignment slots and "M" in the case of model slots. Model slots that contain linking models are connected to the model slots containing the linked models via dashed arrows for visual clarity. Operation nodes are visualized as rectangles with rounded corners, with query nodes such as model properties or model consistency checks labelled "Q", transformation nodes such as model transformations and model synchronizations labelled "T", and mixed nodes such as sub-eGDNs labelled "X" in the top right corner. In addition, all nodes are labeled according to the schema <*name*>:<*type*>.

Figure 4.2 shows an example eGDN that consists of three slot nodes and two operation nodes and realizes a simple chain of model operations. A class diagram stored in the leftmost slot is transformed into an abstract syntax graph via a trans-

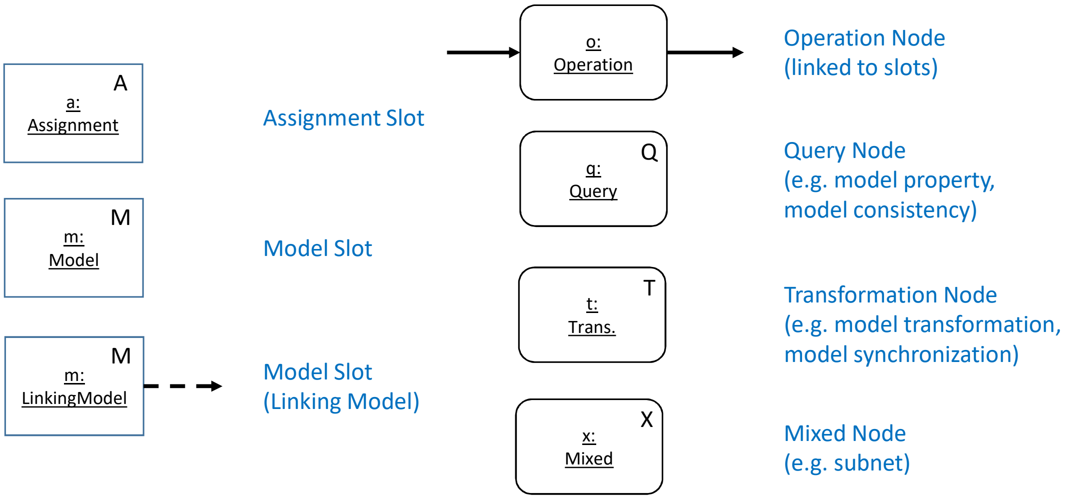

Figure 4.1: Graphical notation for eGDNs

formation node. Then, a query node extracts some information from the created abstract syntax graph and makes the query result accessible via an assignment slot.

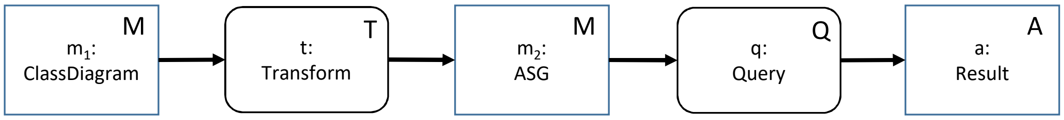

Figure 4.2: Simple example eGDN

5 Incremental Execution of Extended Generalized Discrimination Networks

In this chapter, we describe how eGDNs can be executed to restore consistency in a network of models and model operations in reaction to external changes.

5.1 Definitions regarding Incremental Execution

As a result of edit operations by a user, a model M in the model slot of an eGDN can undergo changes. In this context, a change corresponds to the creation or deletion of a vertex or an edge and is characterized by an atomic model delta of one of four types:

- δ_+^V is a single-element tuple (v), with v a vertex; applying δ_+^V to M modifies M into $M' = (V^M \cup \{v\}, E^M, s^M, t^M)$
- δ_-^V is a single-element tuple (v), with $v \in V^M$, applying δ_-^V to M modifies M into $M' = (V^M \setminus \{v\}, E^M, s^M, t^M)$
- δ_+^E is a tuple (e, s, t), with e an edge and $s, t \in V^M$; applying δ_+^E to M modifies M into $M' = (V^M, E^M \cup \{e\}, s^M \cup \{(e, s)\}, t^M \cup \{(e, t)\})$
- δ_-^E is a single-element tuple (e), with $e \in E^M$; applying δ_-^E to M modifies M into $M' = (V^M, E^M \setminus \{e\}, s^M \setminus \{(e, s^M(e))\}, t^M \setminus \{(e, t^M(e))\})$

Importantly, this notion of atomic deltas can also cover the case of changes to attribute values in models in the form of typed attributed graphs [45]. In this context, attributes can be modeled via dedicated vertices representing attribute values and edges representing the assignment of these values to attributes of regular vertices.

Note that we do not allow implicit deletion of edges. If a vertex is deleted, it must not have any adjacent edges, that is, all adjacent edges have to be deleted previously. Similarly, if an edge is created, adjacent vertices have to be present in the model already.

Changes to the assignment set A in a slot s can similarly be described by atomic slot deltas:

- δ_+^A is a single-element tuple (a), where $a \in dom(s)$; applying δ_+^A to the assignment set A modifies A into $A' = A \cup \{a\}$

- δ^A_- is a single-element tuple (a), where $a \in dom(s)$; applying δ^A_- to the assignment set A modifies A into $A' = A \setminus \{a\}$

For a slot node s, we denote the set of all possible atomic deltas over $dom(s)$ by $dom_\Delta(s)$ and the set of all possible sequences of elements in $dom_\Delta(s)$ by $\mathbb{S}(dom_\Delta(s))$.

Atomic deltas can be applied to a model or assignment set via an *apply* procedure. We overload this procedure to also work with a sequence of atomic deltas, in which case the procedure applies the individual deltas in the order specified by the sequence.

We say that a sequence of atomic deltas Δ is *minimal* for the contents of a slot node s, iff for all possible contents $v \in dom(s)$, it holds that $\nexists \Delta' \in \mathbb{S}(dom_\Delta(s)) : apply(v, \Delta') = apply(v, \Delta)$, where we only consider equality of graphs up to isomorphism.

To enable reacting to model changes with an eGDN $G = (O, S, E, s, t)$, an operation node $o \in O$ with input slots $in(o) = \{s_{i_1}, ..., s_{i_k}\}$ and output slots $out(o) = \{s_{o_1}, ..., s_{o_l}\}$ can be equipped with an *update* procedure. This procedure is parametrized with a valuation function for G and realizes a function $\gamma^\delta : dom(s_{i_1}) \times ... \times dom(s_{i_k}) \times \mathbb{S}(dom_\Delta(s_{i_1})) \times ... \times \mathbb{S}(dom_\Delta(s_{i_k})) \times dom(s_{o_1}) \times ... \times dom(s_{o_l}) \rightarrow \mathbb{F}_\Delta$, with $\mathbb{F}_\Delta$ the set of functions $f_\Delta : out(o) \rightarrow \bigcup_{s_{o_i} \in out(o)} \mathbb{S}(dom_\Delta(s_{o_i}))$ such that $\forall s_{o_i} \in out(o) : f_\Delta(s_{o_i}) \in \mathbb{S}(dom_\Delta(s_{o_i}))$.

To store deltas to react to later, o is also extended by an array $o.\Delta$ that caches sequences of atomic deltas for its input and output slots, which can in practice be collected via a notification mechanism and the observer design pattern [36].

Calling $o.update(val)$ with val a valuation function for G's slots then yields the value of γ^δ parametrized according to val and the cached sequences of deltas, that is, $o.update(val) = \gamma^\delta(val(s_{i_1}), ..., val(s_{i_k}), o.\Delta[s_{i_1}], ..., o.\Delta[s_{i_k}], val(s_{o_1}), ..., val(s_{o_l}))$.

Intuitively, the *update* procedure of an operation node should produce a sequence of deltas for the node's output slots that update the contents of these output slots to be consistent with the current contents of the operation node's input slots. Therefore, in addition to the contents of slots adajcent to o, an *update* procedure may also consider additional information in the form of deltas to input slots to enable a more efficient realization.

Formally, an update procedure *update* of operation node o with input slots $in(o) = \{s_{i_1}, ..., s_{i_k}\}$, output slots $out(o) = \{s_{o_1}, ..., s_{o_l}\}$, and associated function γ^δ is correct iff for parameters $\Delta_1 \in \mathbb{S}(dom_\Delta(s_{i_1})), ..., \Delta_k \in \mathbb{S}(dom_\Delta(s_{i_k}))$, $v_{i_1} \in dom(s_{i_1}), ..., v_{i_k} \in dom(s_{i_k})$, and $v_{o_1} \in dom(s_{o_1}), ..., v_l \in dom(s_{o_l})$,

$$\exists f \in \gamma_S(v_{i_1}, ..., v_{i_k}) : \forall s_{o_i} \in out(o) : apply(v_{o_i}, f_\Delta(s_o)) = f(s_{o_i}),$$

with $f_\Delta = \gamma^\delta(\Delta_1, ..., \Delta_k, v_{i_1}, ..., v_{i_k}, v_{o_1}, ..., v_{o_l})$ and $\forall i \in [1, k], j \in [1, l] : s_{i_i} = s_{o_j} \rightarrow v_{i_i} = v_{o_j}$.

In many cases, the efficient realization of an *update* procedure requires a relaxed notion of correctness, which requires the contents of the output slot to be consistent with the contents of the input slots before the application of the deltas according to o's semantics function. In the following, we will refer to this relaxed notion of correctness as *conditional correctness*, which is formally given by

$$\begin{aligned}
&\exists v'_{i_1} \in dom(s_{i_1}), ..., v'_{i_k} \in dom(s_{i_k}) : \\
&\quad (apply(v'_{i_1}, \Delta_1) = v_{i_1} \wedge ... \wedge apply(v'_{i_k}, \Delta_k) = v_{i_k} \wedge \\
&\quad \exists f' \in \gamma_S(v'_{i_1}, ..., v'_{i_k}) : \\
&\qquad (\forall s_{i_i} \in in(o) \cap out(o) : v'_{i_i} = f'(s_{i_i}) \wedge \\
&\qquad \forall s_{o_i} \in out(o) \setminus in(o) : v_{o_i} = f'(s_{o_i}))) \\
&\rightarrow \\
&\exists f \in \gamma_S(v_{i_1}, ..., v_{i_k}) : \forall s_{o_i} \in out(o) : apply(v_{o_i}, f_\Delta(s_o)) = f(s_{o_i}),
\end{aligned}$$

with $f_\Delta = \gamma^\delta(\Delta_1, ..., \Delta_k, v_{i_1}, ..., v_{i_k}, v_{o_1}, ..., v_{o_l})$ and $\forall i \in [1,k], j \in [1,l] : s_{i_i} = s_{o_j} \rightarrow v_{i_i} = v_{o_j}$.

We say that the realization of an *update* procedure of an operation node o is *fully incremental* iff for a valuation function val and cached deltas $\Delta_1, ..., \Delta_k$ with $\sum_{i \in \{1,...,k\}} |\Delta_i| = 1$, that is, a single atomic delta as an input, (i) the realization's runtime complexity is in $O(|\Delta_o|)$, with $\Delta_o = \bigcup_{s_o \in out(o)} o.update(val)(s_o)$, and (ii) the produced sets of deltas for each output slot are minimal.

In some cases, as with bidirectional or in-place model transformations, operation nodes may be connected to a slot via both an incoming and an outgoing edge, making such a slot simultaneously an input and output slot to the same operation node. Such an operation node may as a result exhibit recursive behavior, since an application of its *update* procedure can also change the contents of the operation node's input slots and thus necessitate further calls to *update* to restore consistency. In this context, we call an *update* procedure of an operation node o is *non-recursive*, if, after one execution of o's update function and subsequent application of the resulting deltas to o's output slot values, a second execution with updated slot values never yields any new deltas.

Formally, an *update* procedure of an operation node o with input slots $in(o) = \{s_{i_1}, ..., s_{i_k}\}$ and output slots $out(o) = \{s_{o_1}, ..., s_{o_l}\}$, is *non-recursive*, if for any possible parametrization $v_{i_1} \in dom(s_{i_1}), ..., v_{i_k} \in dom(s_{i_k})$, $\Delta_1 \in \mathbb{S}(dom_\Delta(s_{i_1})), ..., \Delta_k \in \mathbb{S}(dom_\Delta(s_{i_k}))$, and $v_{o_1} \in dom(s_{o_1}), ..., v_{o_l} \in dom(s_{o_l})$, it holds that

$$\forall s_o \in out(o) : \gamma^\delta(\Delta'_1, ..., \Delta'_k, v'_{i_1}, ..., v'_{i_k}, v'_{o_1}, ..., v'_{o_l})(s_o) = \emptyset,$$

where

$$\Delta'_i = \begin{cases} f_\Delta(s_i) & \text{if } s_{i_i} \in out(o) \\ \Delta_i & \text{otherwise} \end{cases}$$

and

$$v'_{i_i} = \begin{cases} apply(v_{i_i}, f_\Delta(s_i)) & \text{if } s_{i_i} \in out(o) \\ v_{i_i} & \text{otherwise} \end{cases}$$

and

$$v'_{o_i} = apply(v_{o_i}, f_\Delta(s_{o_1})),$$

with $f_\Delta = \gamma^\delta(\Delta_1, ..., \Delta_k, v_{i_1}, ..., v_{i_k}, v_{o_1}, ..., v_{o_l})$.

The *potential update directions* of an update procedure of operation node o for a set of input slots $S_i \subseteq in(o)$ are given by $o.dir_\Delta(o, S_i)$, where for a slot $s_o \in out(o)$,

$$\begin{aligned} s_o \in o.dir_\Delta(o, S_i) \leftrightarrow & \exists \Delta_1 \in \mathbb{S}(dom_\Delta(s_{i_1})), ..., \Delta_k \in \mathbb{S}(dom_\Delta(s_{i_k})), \\ & v_{i_1} \in dom(s_{i_1}), ..., v_{i_k} \in dom(s_{i_k}), \\ & v_{o_1} \in dom(s_{o_1}), ..., v_{o_l} \in dom(s_{o_l}) : \\ & \quad \forall s_{i_i} \in in(o) \setminus S_i : \Delta_i = \emptyset \wedge \\ & \quad \gamma^\delta(\Delta_1, ..., \Delta_k, v_{i_1}, ..., v_{i_k}, v_{o_1}, ..., v_{o_l})(s_o) \neq \emptyset \end{aligned}$$

Intuitively, $o.dir_\Delta(o, S_i)$ thus denotes the subset of output slots for which o's update procedure may generate deltas if the contents of at most the input slots in S_i have changed.

A function dir_Δ for potential update directions is monotonic by definition in the sense that $\forall S_{i_1}, S_{i_2} \subseteq in(o) : S_{i_1} \subseteq S_{i_2} \rightarrow o.dir_\Delta(o, S_{i_1}) \subseteq o.dir_\Delta(o, S_{i_2})$. We say that dir_Δ is *union monotonic* if it furthermore holds that $\forall S_{i_1}, S_{i_2} \subseteq in(o) : o.dir_\Delta(S_{i_1}) \cup o.dir_\Delta(S_{i_2}) = o.dir_\Delta(S_{i_1} \cup S_{i_2})$.

In the following, we present algorithms for the incremental execution of an eGDN based on the *update* procedures of its operation nodes. For these algorithms, we assume that deltas cached in the input eGDN are consistent in the sense that they correspond to a modification from slot contents that were consistent with the semantics functions of all operations in the eGDN to the current contents. Intuitively, this assumption simply implies that the presented algorithms can only produce consistent slot contents if the slot contents were previously consistent at some point and all changes since then have been tracked and cached in the eGDN.

5.2 Incremental Execution with Guaranteed Termination

Given a correct *update* function for each operation node, an eGDN $G = (O, S, E, s, t)$ can be executed incrementally in the context of a valuation function *val* via Algorithm 1. Therefore, Algorithm 1 first derives an ordering of G's operation nodes and then updates the *val* function by executing the nodes' *update* functions, applying the resulting deltas to the appropriate slots, and updating the cached deltas.

Importantly, the employed ordering has to guarantee correct results in the sense that the contents of G's slots after the execution must be consistent with the semantics functions of all of its operation nodes, that is, it must hold that $\forall o \in O : o.valid(val)$.

If G takes the form of a directed acyclic graph and operation nodes do not share output slots, such an ordering can be obtained by simply sorting G's operation nodes topologically. However, requiring DAG structure represents a substantial restriction, as it effectively prohibits bidirectional transformations where some input slots are also output slots. Moreover, the assumption regarding the complete absence of shared output slots, while required to prevent overwriting of operation's results, is another obstacle to realizing several desirable use cases, for instance those involving chains of bidirectional transformations.

Based on the properties of an eGDN's operation nodes with respect to non-recursiveness and potential update directions, an appropriate order can also be found for certain cyclical eGDNs, with a relaxed assumption regarding shared output slots. Algorithm 2 represents an analysis for an eGDN G that contains only nodes with non-recursive *update* procedures and a set of slots S_i with initially modified contents. If successful, the algorithm returns an execution order that can be used instead of the topological ordering in Algorithm 1. Importantly, the computed ordering still yields a valuation function that is consistent with all operations' semantics.

Algorithm 1: Incremental algorithm for executing an eGDN based on an ordering of its operation nodes

```
Procedure ExecuteIncrementalDAG($G = (O, S, E, s, t), val$)
    Input : $G$: The eGDN
            $val$: A valuation function for $G$'s slots
    $D \leftarrow$ FindValidUpdateOrder($O, \{s \in S | \exists o \in O : o.\Delta[s] \neq \emptyset\}$);
    if $D \neq$ null then
        foreach $o \in D$ do
            $\Delta_o \leftarrow o.update(val)$;
            foreach $s_o \in out(o)$ do
                $val(s_o) \leftarrow$ apply($val(s_o), \Delta_o(s_o)$);
                foreach $o' \in out(s_o)$ do
                    $o'.\Delta[s_o] \cup \Delta_o$;
                end
            end
            foreach $s \in in(s) \cup out(s)$ do
                $o.\Delta[s] \leftarrow \emptyset$;
            end
        end
    end
```

The algorithm first creates an array C with one cell per operation node in O and initializes it with empty sets. It also initializes a queue Q with all operation nodes that are connected to a slot in S_i and, for each such operation node, stores the set of its input slots that are also in S_i in the corresponding cell in C. Then, a slightly modified breadth-first search is performed over the eGDN structure using

Algorithm 2: Static analysis algorithm for finding an eGDN update order

```
Procedure FindValidUpdateOrder($G = (O, S, E, s, t), S_i$)
    Input : $G$: The eGDN
            $S_i$: The set of initially changed slots
    $C \leftarrow$ new $Array(|O|)$;
    $C.init(\emptyset)$;
    $Q \leftarrow$ new $Queue$;
    $G_T =$ new $Graph$;
    foreach $o \in in(S_i) \cup out(S_i)$ do
        $Q.enqueue(o)$;
        $C[o] \leftarrow S_i \cap in(o)$;
    end
    $G_T.addVertices(Q)$;
    while $\neg Q.isEmpty()$ do
        $o \leftarrow Q.dequeue()$;
        $S_o \leftarrow o.dir_\Delta(o, C[o])$;
        $O_o \leftarrow out(S_o) \cup in(S_o) \setminus \{o\}$;
        foreach $o' \in O_o$ do
            if $\neg o' \in Q$ then
                $Q.enqueue(o')$;
            end
            $C[o'] \leftarrow C[o'] \cup (S_o \cap in(o'))$;
            $G_T.addVertexIfNotExists(o')$;
            $G_T.createEdgeIfNotExists(o, o')$;
            if $G_T.hasCycle()$ then
                return null;
            end
        end
        $C[o] \leftarrow \emptyset$;
    end
    return SortTopologically($G_T$);
```

the initialized queue Q to essentially simulate an execution of G without concrete inputs.

Therefore, the procedure loops until Q is empty. In each loop execution, the first operation node o in Q is dequeued. Then, all output slot nodes for which deltas could be produced due to the execution of o's update procedure S_o are obtained based on o's potential update directions and the set of slots that might currently contain unhandled deltas, which is retrieved from C. Afterwards, all operation nodes o' connected to a slot in S_o are added to Q if they are not yet contained. Also, the set of o's input slots with potentially unhandled deltas stored in C is updated based on S_o. An exception is made for the currently considered node o, which is never added to the queue again and whose set of input slots with potentially unhandled deltas is reset to the empty set, exploiting the assumption that all *update* procedures in the eGDN are non-recursive.

During execution, the algorithm keeps track of the dependencies between G's operations in a trigger graph G_T. Execution aborts by returning null as soon as a cyclical dependency is detected, which may indicate a potential infinite loop in G's execution for the initially populated slots S_i. This also guarantees that after a full execution of the loop in line 10, G_T is a DAG.

Finally, a topological ordering of G_T, is returned as a possible canonic execution order that, under the mentioned assumptions, produces a valuation function for the input eGDN's slots that is consistent with the semantics functions of all of the eGDN's operation nodes.

While the presented algorithm is formulated to handle incremental changes to a network of models and model operations, the batch case that requires an initial execution of model operations to derive corresponding query results and transformed models for an initial set of existing models can be handled in a straightforward manner. Therefore, an incremental construction of the initially existing models can be emulated by deriving trivial sequences of corresponding creation operations, which can act as the starting point for the algorithm. This only requires the assumption that the case where all slots of an eGDN are empty constitutes a consistent valuation regarding the semantics of all of the eGDN's operations, which seems reasonable. The additional assumption is essentially required to satisfy the rerquirement regarding consistency of initially cached deltas with the current state.

Termination

By including the additional termination criterion in the loop in line 10 of Algorithm 2 that requires the constructed dependency graph to be acyclic, Algorithm 2 is guaranteed to terminate.

Theorem 1. *Algorithm 2 always terminates.*

Proof. Except for the loop in line 10, all loops only iterate over finite sets, and all individual operations always terminate. The loop in line 10 also always terminates due to the termination criterion regarding cyclical dependencies between the eGDN's operation nodes: Since one operation node is removed from Q in each loop iteration, termination is only threatened if operation nodes keep getting added to Q. Since there is only a finite number of operation nodes, infinite behavior can only occur as a result of cycles in the modified breadth-first search. However, such cycles are detected via G_T and immediately lead to abortion of the execution. $\square$

Consequently, Algorithm 1 is also guaranteed to terminate if the execution of the input eGDN's *update* procedures always terminates.

Theorem 2. *For an input eGDN G, Algorithm 1 always terminates if the update procedures of G's operation nodes always terminate.*

Proof. According to Theorem 1, Algorithm 2 always terminates by either aborting or returning a sequence of operation nodes. Such a sequence being returned implies that the sequence is finite. The loop in line 3 is thus only executed for finitely many iterations. Since all other loops only iterate over finite sets and all individual operations always terminate due to the assumption regarding G's *update* procedures, Algorithm 1 always terminates. $\square$

Correctness

The following theorem states the correctness of a canonic execution order resulting from an execution of Algorithm 2 for the case that all dir_Δ functions are union monotonic.

Theorem 3. *For inputs $G = (O, S, E, s, t)$ and val, if all update procedures in G are correct and non-recursive, all dir_Δ functions in G are union-monotonic, and if the valuation function before the application of the deltas cached in G was consistent with the semantics of G's operation nodes, Algorithm 1 aborts or produces a final valuation function val such that $\forall o \in O : o.valid(val)$.*

Proof. If Algorithm 1 does not abort, a canonic execution order R for G's operation nodes has been generated by topologically sorting the resulting directed acyclic dependency graph G_T of a terminating execution of Algorithm 2.

Due to the non-recursiveness of G's operation nodes, we know that after executing an operation node o via Algorithm 1, it holds that $o.valid(val)$. Thus, for an operation node o, $\neg o.valid(val)$ can only hold after executing the entire sequence R if there exists some operation node o' that comes after o in R and that changes the contents of a slot adjacent to o or if $o \notin R$. Considering that all operation nodes that have an adjacent slot with initially modified contents are initially added to G_T, the algorithm has terminated, and that prior to the cached modifications of G's slots, slot contents were consistent with all operation nodes' semantics, for a node $o \notin R$, $\neg o.valid(val)$ can also only hold if there is some operation node $o' \in R$ that changes the contents of a slot s adjacent to o.

In either case, we know that there cannot exist an edge from o' to o in G_T, because o' either comes after o in the topological ordering or because the addition of such an edge would have caused o to be added to G_T and consequently R. This means that, due to the definition of $o'.dir_\Delta$ and because of the assumed union monotonicity, there must be a slot s' with $o'.\Delta[s] \neq \emptyset$ and $s \in o'.dir_\Delta(\{s'\})$ before executing o' that was never in the set of slots $C[o']$ when o' was dequeued in line 11 of Algorithm 2. Since slots are only removed from $C[o']$ when o' is dequeued and corresponding edges are added, we know that $s' \notin S_i$ and thus $o'.\Delta[s'] = \emptyset$ at the start of Algorithm 1, as otherwise, o' would have been added to Q and the edge between o' and o would eventually have been created.

There hence must be a node o'' that comes before o' in R that modified the contents of s'. Also for o'', there must be a slot s'' with $o''.\Delta[s''] \neq \emptyset$ and $s' \in o''.dir_\Delta(\{s''\})$ before executing o'' that was never in $C[o'']$ whenever o'' was dequeued (because otherwise, the edge between o' and o would have been created eventually). Therefore, again, there must be an operation node before o'' in R that modified the contents of s'' and for which the same constraints apply as for o''. Ultimately, this implies that for the first operation node in the sequence, there must be a predecessor that changes the contents of some slot node, which is obviously a contradiction.

Hence, there cannot be an operation node in R whose execution changes the contents of a slot adjacent to a previous operation node in R or an operation node not contained in R. Consequently, we know that after executing R, $\forall o \in O : o.valid(val)$. ☐

If the eGDN's *update* functions are only conditionally correct, an additional constraint has to be introduced regarding eGDN structure to guarantee correctness. Namely, operation nodes may not share output slots if the output slots are not also input slots of all sharing operation nodes, and output slots of a node that are not simultaneously input slots of the same node may not have their contents modified by users.

Intuitively, these conditions impose the restriction on the eGDN structure that the contents of an operation node's output slot may not be modified by another operation node or a user without that operation node being able to pick up on and handle the changes.

Corollary 1. *Assuming that* $\forall o_1, o_2 \in O : o_1 \neq o_2 \rightarrow \forall s \in out(o_1) \cap out(o_2) : s \in in(o_1) \wedge s \in in(o_2)$ *and* $\forall o \in O : \forall s \in out(o) : o.\Delta[s] = \emptyset$*, for inputs* $G = (O, S, E, s, t)$ *and* val*, if all update procedures in* G *are conditionally correct and non-recursive, all* dir_Δ *functions in* G *are union-monotonic, and if the valuation function before the deltas cached in* G *was consistent with the semantics of* G*'s operation nodes, Algorithm 1 aborts or produces a final valuation function* val *such that* $\forall o \in O : o.valid(val)$.

Proof. From the additional assumptions regarding output slots of G's operation nodes it follows directly that the condition in the definition of conditional correctness is never violated. Thus, the statement from Theorem 3 also applies for the case of conditionally correct *update* functions. ☐

Notably, the order in which operation nodes are added to the queue Q in lines 5 and 14 of Algorithm 2 is undefined. Since the order of operation nodes in Q affects the behavior of the algorithm, this might mean that Algorithm 2 is ultimately not deterministic.

We can however show that, if Algorithm 2 does not abort due to cycles in G_T, the final dependency graph G_T is uniquely defined, independently of the order in which operation nodes are added to Q. Thus, the only remaining nondeterminism in Algorithm 2 affecting the result stems from the topological sorting at the end of the algorithm, which is an inherently nondeterministic operation.

Theorem 4. *For inputs $G = (O, S, E, s, t)$ and S_i, the dependency graph G_T after a full execution of the loop in line 10 of Algorithm 2 is uniquely defined up to isomorphism if all dir_Δ functions in G are union monotonic.*

Proof. The set of vertices initially added to G_T is uniquely determined by $in(S_i) \cup out(S_i)$. Since additional vertices are only ever added in conjunction with the creation of an edge, the set of vertices added during the execution of the loop in line 10 is determined by the set of added edges.

To show the unique determination of added edges by the algorithm's inputs, we show that in a terminating execution of the loop, the initial set S_i in conjunction with the eGDN G uniquely determines a set of pairs of operation nodes (o_1, o_2), between which directed edges are created in G_T.

S_i uniquely determines the set of operation nodes $O_Q = in(S_i) \cup out(S_i)$ that is initially added to Q. For each of these operation nodes $o_Q \in O_Q$, due to the monotonicity of $o_Q.dir_\Delta$ and because slots are only removed from $C[o_Q]$ after o_Q has been dequeued and processed, at least the edges for pairs $edges_S(o_Q, S_i) = \{(o_Q, o_T) | o_T \in out(S_o) \cup in(S_o) \setminus \{o_Q\}\}$ are added to G_T, where $S_o = o_Q.dir_\Delta(S_i \cap in(o_Q))$ when o_Q is dequeued. Due to the assumption regarding union monotonicity, we can also write $edges_S(o_Q, S_i) = edges_\emptyset(o_Q) \cup \bigcup_{s_i \in S_i \cap in(o_Q)} edges_N(o_Q, s_i)$, with $edges_\emptyset(o_Q) = \{(o_Q, o_T) | o_T \in out(o_Q.dir_\Delta(\emptyset)) \cup in(o_Q.dir_\Delta(\emptyset)) \setminus \{o_Q\}\}$ and $edges_N(o_Q, s_i) = \{(o_Q, o_T) | o_T \in out(o_Q.dir_\Delta(in(o_Q) \cap \{s_i\})) \cup in(o_Q.dir_\Delta(in(o_Q) \cap \{s_i\})) \setminus \{o_Q\}\}$.

In addition, the modification of C and Q that takes place for each dequeued $o_Q \in O_Q$ may cause the addition of further edges down the line. Specifically, for each $s_o \in o_Q.dir_\Delta(\{s_i\})$ and each $o_T \in out(s_o) \cup in(s_o) \setminus \{o_Q\}$, o_T, if not already contained, is added to Q and subsequently handled in the same way as o_Q, with s_i guaranteed to be in $C[o_T]$ at that moment. This will cause the addition of all edges corresponding to the pairs $edges_N(o_T, s_o)$ and again trigger the addition of further edges. Due to the monotonicity of $o_T.dir_\Delta$ and because slots are only removed from $C[o_T]$ when o_T is dequeued, the addition of these edges happens independently from any other modifications to $C[o_T]$ that might be made in the meantime. Furthermore, due to the assumption regarding union monotonicity of $o_T.dir_\Delta$, a combination of modifications of $C[o_T]$ cannot yield any additional edges compared to what is yielded for the individual members of $C[o_T]$.

Because neither can $C[o_T]$ be modified in any other way, nor can edges be added to G_T in any other way, the set of pairs of operation nodes (o_1, o_2) between which directed edges are created in G_T in the loop is given by the function $edges(S_i) = \bigcup_{o_Q \in in(S_i) \cup out(S_i)} (edges_\emptyset(o_Q) \cup \bigcup_{s_i \in S_i \cap in(o_Q)} edges_R(o_Q, s_i))$, where $edges_R(o_Q, s_i) = edges_N(o_Q, s_i) \cup \bigcup_{o_T \in O_T)) \setminus \{o_Q\}} \bigcup_{s_o \in o_Q.dir_\Delta(in(o_Q) \cap \{s_i\}} edges_R(o_T, s_o)$, where O_T is given by $O_T = out(o_Q.dir_\Delta(in(o_Q) \cap \{s_i\})) \cup in(o_Q.dir_\Delta(in(o_Q) \cap \{s_i\}$.

The loop terminating due to Q becoming empty implies that all nodes ever added to Q have been processed and hence all corresponding edges have been added to G_T. Since it is ensured that for each pair of operation nodes (o_1, o_2), only one corresponding edge is added, we know that regardless of the concrete processing order, G_T always contains exactly one directed edge for each pair $(o_1, o_2) \in edges(S_i)$.

Since the set of added vertices is uniquely determined by the set of added edges and each vertex can only be added once, the set of G_T's vertices is uniquely defined for inputs G and S_i.

The graph G_T at the end of a full execution of the loop in line 10 of Algorithm 2 is hence uniquely defined for inputs G and S_i, regardless of the order in which operation nodes are added to Q in lines 5 and 14. ☐

The fact that G_T is uniquely defined by the inputs G and S_i also implies that if an execution of Algorithm 1 terminates without aborting, so does any possible execution for the same inputs.

Theorem 5. *An execution of the loop in line 10 of Algorithm 1 terminates without aborting for inputs $G = (O, S, E, s, t)$ and S_i if and only if any other execution for the same inputs also terminates without aborting.*

Proof. According to Theorem 1, the loop in line 10 of Algorithm 1 always terminates, either because of a violation of the looping condition or because the loop aborts. Since the loop aborts if and only if a cycle is detected in G_T at any point and edges are never removed from G_T, it follows that the loop terminates without aborting if and only if the set of edges added to G_T during the loop execution does not form cycles. Since the set of edges added to G_T during the loop execution is functionally determined by only the inputs G and S_i, it hence follows that, if an execution of the loop terminates without aborting for G and S_i, any execution with the same inputs will also terminate without aborting. ☐

Furthermore, we can show that if there exists an execution sequence for G that guarantees correct results in the worst case and that executes every operation node at most once, Algorithm 2 finds such a sequence.

Theorem 6. *For an input eGDN $G = (O, S, E, s, t)$ with correct and non-recursive update procedures with union monotonic dir_Δ functions and a set of slots $S_i \subseteq S$ with initially modified contents for a valuation function val, assuming that*

1. *for any operation node $o_1 \in O$, for any execution of $o_1.update(val')$ with a valuation function val' and deltas for input slots S_Δ, it holds that $\forall s_o \in o_1.dir_\Delta(S_\Delta) : o_1.update(val')(s_o) \neq \emptyset$,*
2. *for a second node $o_2 \in O$ with $o_1 \neq o_2$, it holds that $\exists s_o \in out(o_1) \cap in(o_2) : o_1.update(val')(s_o) \neq \emptyset \to \neg o_2.valid(val'')$, where for $s \in S$*

$$val''(s) = \begin{cases} apply(val'(s), o_1.update(val')(s)) & \text{if } s \in out(o_1) \cap in(o_2) \\ val'(s) & \text{otherwise} \end{cases} \tag{5.1}$$

and

3. *it holds that* $\forall o \in in(S_i) \cup out(S_i) : \neg o.valid(val)$,

if there exists a sequence that guarantees a correct resulting valuation function if executed via Algorithm 1 and that only contains each node $o \in O$ *once, Algorithm 2 returns such a sequence.*

Proof. Under the given assumptions, the set of edges in G_T created by Algorithm before termination or abortion represents a subset of all relations between pairs of operation nodes (o_1, o_2), where o_1's $update$ procedure has to be executed at least once to produce a correct final valuation function and that execution modifies the contents of a slot adjacent to o_2, necessitating the subsequent execution of o_2 according to assumption (2).

This is due to the fact that, to restore consistency, all operation nodes in $o \in in(S_i) \cup out(S_i)$ have to be executed at least once according to assumptions (2) and (3). All these operation nodes o_1 are initially added to the queue Q in Algorithm 2. Each execution of an operation node o_1, according to assumption (1), modifies all slots in $o_1.dir_\Delta(S_i \cap in(o_1))$, which necessitates a subsequent execution of all operation nodes $o_2 \in in(o_1.dir_\Delta(S_i \cap in(o_1))) \cup out(o_1.dir_\Delta(S_i \cap in(o_1)))$ according to assumption (2). Algorithm 2 creates edges for all these pairs (o_1, o_2) when o_1 is dequeued.

The subsequent execution of any operation node o_2 similarly necessitates the execution of all nodes $o_3 \in in(o_2.dir_\Delta(S_i \cap in(o_2))) \cup out(o_2.dir_\Delta(S_i \cap in(o_2)))$, which is also reflected by the edges created in Algorithm 2 when o_2 is dequeued, and so on. Since the algorithm creates no additional edges due to the assumption regarding union monotonicity of the dir_Δ functions, all edges in G_T represent such necessary relationships on the ordering of operation nodes[1].

Since Algorithm 2 always produces a correct sequence of operation nodes if G_T is acyclic, we can assume that in the case where the algorithm does not produce an ordering, there is at least one cycle in G_T. There hence cannot exist a sequence of the operation nodes involved in this cycle where each node is only contained once and each node is executed at least once after its predecessor in the cycle. Thus, by contraposition it follows that, if there exists a sequence of operation nodes that guarantees correct results and where each operation node is only contained once, Algorithm 2 finds such a sequence. ☐

Note that there may be finite orders of operation node executions that guarantee correct results based on the assumptions in Theorem 6 that are not found by Algorithm 2. However, these orders require that at least one operation node is executed at least twice.

---

[1] As a side note, since operation nodes can be dequeued/executed with different sets of potentially modified input slots, an edge between nodes (o_1, o_2) in G_T does not necessarily mean that o_2 has to be executed after *any* execution of o_1, but only that such a subsequent execution is necessary *at least once*.

5.3 Incremental Execution of Arbitrary eGDNs

If the eGDN is not a DAG and no suitable ordering of its operation nodes can be found via Algorithm 2, incremental execution can instead be achieved via a simple fixpoint iteration as in Algorithm 3.

Algorithm 3: Incremental algorithm for eGDN execution

```
Procedure ExecuteIncremental($G = (O, S, E, s, t), val$)
    Input : $G$: The eGDN
            $val$: A valuation function for G's slots
    $D \leftarrow \{o \in O | \exists s \in in(o) \cup out(o) : o.\Delta[s] \neq \emptyset\}$;
    while $D \neq \emptyset$ do
        $D_n \leftarrow \emptyset$;
        foreach $o \in D$ do
            $D \leftarrow D \setminus \{o\}$;
            $\Delta_o \leftarrow o.update(val)$;
            foreach $s \in in(s) \cup out(s)$ do
                $o.\Delta[s] \leftarrow \emptyset$;
            end
            foreach $s_o \in out(o)$ do
                if $\Delta_o(s_o) \neq \emptyset$ then
                    $val(s_o) \leftarrow \texttt{apply}(val(s_o), \Delta_o(s_o))$;
                    foreach $o' \in out(s_o)$ do
                        $o'.\Delta[s_o] \cup \Delta_o$;
                        if $o' \notin D$ then
                            $D_n \leftarrow D_n \cup \{o'\}$;
                        end
                    end
                    foreach $o' \in in(s_o)$ do
                        if $o' \neq o \wedge o' \notin D$ then
                            $D_n \leftarrow D_n \cup \{o'\}$;
                        end
                    end
                end
            end
        end
        $D \leftarrow D_n$;
    end
```

Algorithm 3 first initializes the set of operation nodes that require execution D with the set of all operation nodes in the input eGDN for which there are changes in one of the node's input or output slots. Then, the algorithm iterates until a fixpoint is reached.

Therefore, a set of operation nodes that will require execution in the next iteration D_n is initialized with the empty set. Afterwards, for each operation node o that is due for execution in the current iteration, that node is removed from the set D. Then, o's *update* procedure is called to compute a set of changes to the contents of o's output slots to make them consistent with the semantics of o.

For each output slot s_o of o that *update* has computed changes for, these changes are subsequently applied and appropriately registered at each operation node o' for which s_o is an input slot. If any such o' is not still due for execution in the current iteration, it is marked for execution in the next iteration by adding it to D_n. Operation nodes for which s_o is an output slot are similarly marked for execution. Finally, after all operation nodes in D have been considered, D_n replaces D and a new iteration starts if D_n is not empty.

Analogously to Algorithm 1, Algorithm 3 can handle the batch case of an initial eGDN execution for existing models by encoding such existing models as sequences of element creations.

Termination

In contrast to Algorithm 1, Algorithm 3 is not guaranteed to terminate, since cyclical transitive dependencies of operation nodes may cause infinite cycles of changes to the contents of some slot node. Without restricting developers in what kinds of eGDNs they are allowed to specify, this problem is inevitable.

In practice however, termination of networks of model operations like eGDNs can be achieved despite the presence of cyclical structures. In some cases for instance, cycles at the network level do not necessarily correspond to actual cyclical dependencies of model operations if the involved model operations only affect distinct parts of slot contents, such as elements of certain, distinct types. In some cases, a restructuring of the eGDN may remove cycles at the structural level while preserving semantics, for instance by converting in-place model transformations without an effective reflexive dependency into a model transformation with distinct input and output models.

Moreover, cycles of model operations may exhibit monotonic behavior, for instance by deleting certain elements in each iteration that are never recreated, thus guaranteeing convergence. Ultimately however, it remains the responsibility of the developers to create networks of model operations that do not lead to infinite loops in execution.

Correctness

If Algorithm 3 terminates, the resulting valuation function is guaranteed to be consistent with the semantics of all operation nodes in the input eGDN.

Theorem 7. *For inputs $G = (O, S, E, s, t)$ and val, if Algorithm 3 terminates, all employed $update$ procedures are correct and non-recursive, and if the valuation function before the ap-*

plication of the deltas cached in G was consistent with the semantics of G's operation nodes, the algorithm produces a final valuation function val such that $\forall o \in O : o.valid(val)$.

Proof. We show that the invariant (1) $\forall o \in O : \neg o.valid(val) \rightarrow o \in D$ holds for the loop in line 2 via induction over the number of loop iterations.

The base case for invariant (1) holds due to the initialization of D and the assumption regarding the initial cached deltas and previous valuation function.

To show the induction step for invariant (1), we first show that under the induction assumption, the invariant (2) $\forall o \in O : \neg o.valid(val) \rightarrow o \in D \cup D_n$ holds for the loop in line 10. This can also be done via induction.

The base case for invariant (2) holds due to the induction assumption of (1).

The induction step holds for invariant (2) since in each iteration of the inner loop, only one operation node o is executed via its $update$ procedure and removed from D, updating val and the cached deltas in the process. If the execution of o does not change the contents of one of its own input slots, we know that afterwards, $o.valid(val)$ due to the assumption regarding correctness of $update$ procedures and because the cached deltas are always updated correctly. Otherwise, o is added to D_n in the loop in line 13. The loops in line 13 and 19 also add all operation nodes o' to D_n for which the result of $o'.valid(val)$ may have been impacted by the update to val. Thus, given the induction assumption, at the end of the loop in line 10, we again have $\forall o \in O : \neg o.valid(val) \rightarrow o \in D \cup D_n$ and hence the induction step holds.

Since at the end of the loop in line 10, $D = \emptyset$, we know that $\forall o \in O : \neg o.valid(val) \rightarrow o \in D_n$. Because at the end of the iteration of the loop in line 2, the set D is replaced by D_n, the induction step for (1) holds.

Since the loop in line 2 is only left when $D = \emptyset$ after the replacement with D_n, we know that, if the algorithm terminates, $\forall o \in O : o.valid(val)$. □

Similar to Algorithm 1, the algorithm also yields correct results if all employed $update$ procedures are at least conditionally correct, the eGDN's nodes do not share output slots that are not also input slots to all sharing nodes, and there are no deltas for an output slot of a node that is not simultaneously an input slot.

Corollary 2. *Assuming that* $\forall o_1, o_2 \in O : o_1 \neq o_2 \rightarrow \forall s \in out(o_1) \cap out(o_2) : s \in in(o_1) \wedge s \in in(o_2)$ *and* $\forall o \in O : \forall s \in out(o) : o.\Delta[s] = \emptyset$*, for inputs* $G = (O, S, E, s, t)$ *and* val*, if Algorithm 3 terminates, all employed* $update$ *procedures are conditionally correct and non-recursive, and if the valuation function before the deltas cached in G was consistent with the semantics of G's operation nodes, it produces a final valuation function val such that* $\forall o \in O : o.valid(val)$.

Proof. From the additional assumptions regarding output slots of G's operation nodes, it follows directly that the condition in the definition of conditional correctness is never violated. Thus, the statement from Theorem 7 also applies for the case of conditionally correct $update$ functions. □

5.4 Development with eGDNs

Since Algorithm 2 considers only the eGDN structure and no concrete slot contents, it can be employed as a tool for statically analyzing eGDNs. In particular, via the algorithm, configurations of slots with modified contents can be analyzed regarding termination of a corresponding eGDN execution. For instance, the algorithm can be used to check whether termination is guaranteed if a specific individual model is modified.

If this is the case for all user-editable models, a conservative approach that always guarantees terminating eGDN executions and correct results while avoiding the exponential effort of executing the analysis for every combination of user-editable models would be enforcing a *direct propagation policy*. Under this policy, after modifying a single model, the corresponding changes would immediately be propagated to restore consistency. Only after that, the modification of a different model would be permitted.

Furthermore, Algorithm 2 can be adapted to return the set of slots $closure_{\Delta}(S_i)$ that may be automatically modified by eGDN operations if the eGDN were to be executed via Algorithm 1 with initially modified slots S_i. This enables collaborative development of a network of models managed via an eGDN with guaranteed termination and conflict-free consistency restoration via a *propagation closure locking policy*. For a set of already modified slots S_{Δ}, this policy would only allow modification of the contents of another slot s if, for the set $S_{\Delta} \cup \{s\}$, Algorithm 2 produces an execution order. Furthermore, to guarantee that no user edits are overwritten, the policy would check whether $S_{\Delta} \cup \{s\} \cap closure_{\Delta}(S_{\Delta} \cup \{s\}) = \emptyset$. Note that the restrictions of this policy would also apply in the case where the same user wants to edit the contents of multiple slots.

Since Algorithm 3 does not guarantee termination, careful consideration is required if an eGDN cannot be executed via Algorithm 1. However, if developers are confident that their eGDN is guaranteed to terminate despite cyclical dependencies at the structural level, Algorithm 3 can be used as a fallback option for eGDN execution.

The presented algorithms also enable the treatment of sub-eGDNs as operation nodes of a parent eGDN, as they essentially provide a realization of the required *update* procedure.

6 Implementation

We have prototypically implemented a number of concrete example operation node types for the construction of eGDNs for usage in the context of the Eclipse Modeling Framework (EMF) [26]. In addition to listing the implemented operations' names, Table 6.1 also provides brief descriptions of their behavior. Table 6.2 characterizes our implementations in terms of the properties defined in this report.

Non-recursiveness: The *update* procedure of TGG Snychronisation operations is non-recursive if the slots containing source, target, and correspondence model are distinct. The non-recursiveness of composite nodes depends on the exact composition of the sub-eGDN. All other nodes' *update* procedures are only guaranteed to be non-recursive if their input and output slots are distinct. The checkmark symbol ✓indicates non-recursive *update* procedures under this assumption.

Potential Population/Update Directions: The potential update directions of the TGG Synchronization ($\leftrightarrow$) can be characterized as follows (under the assumption of distinct slots for source, target, and correspondence model): If the set of considered input slots is empty, no modifications will be made to the contents of any output slot. If the set of considered input slots contains only the source model, the operation will only modify the target model and correspondence model and vice-versa. In all other cases, all models may be modified. The potential update directions of composite nodes are determined by the exact structure of the sub-eGDN. All other nodes may modify the contents of all of their output slots for any set of considered input slots. The potential update direction function dir_Δ of all example nodes is union monotonic.

Correctness: The *update* procedures of all operation implementations are only conditionally correct. Effectively, this means that operations may not share output slots and no user edits are allowed to output slots of operation nodes, unless the shared or edited output slot is also simultaneously an input slot of the concerned operation nodes.

Incrementality: The checkmark symbol ✓indicates a fully incremental *update* procedure under the assumption of ideal data structures. Also operations which are listed as not fully incremental support incremental execution to some extent. The degree of incrementality depends on the operation and its concrete inputs. Naturally, our implementation of the Expression node is only fully incremental if the evaluation of the considered expression has a runtime complexity in $O(1)$. In the case of the Pattern Matching and TGG Synchronization node, a fully incremental execution can be achieved for certain input models and patterns respectively TGGs. The degree of incrementality of the execution of an eGDN or sub-eGDN depends on which slots are designated the eGDN's interface slots, as well as the contained operation nodes and their composition. While the Group Expression node also has a partially incremental update procedure, due to the handling of collections via the employed OCL-interpreter, a fully incremental execution is usually not possible.

As interfaces between these operations, that is, slot nodes, our implementation employs regular EMF models for model slots and hash-based indices for assignment slots. While the choice of hash-based indices over array-based indices means that the theoretically fully incremental operation implementations may not be fully incremental in conjunction with our slot implementations, hash-based data structures are usually preferable in practice due to their lower memory footprint and exhibit acceptable performance in most scenarios.

Figure 6.1 shows a more complex version of the example eGDN from Figure 4.2 that can be realized using the introduced example eGDN nodes from Table 6.1. The transformation from class diagram to abstract syntax graph is now concretely realized via a unidirectional TGG Synchronization. The query operation that was previously represented by a single query node is decomposed into a complex network of subqueries. This sub-eGDN consists of two Pattern Matching nodes labeled "$x \rightarrow y$" that look for primitive patterns consisting of a single edge, one Group Count and one Group Sum node visualized as nodes labeled "COUNT (X)" respectively "SUM (X)", and a Join node labeled "⋈". Alternatively, the Pattern Matching nodes could also be realized as Edge Inputs.

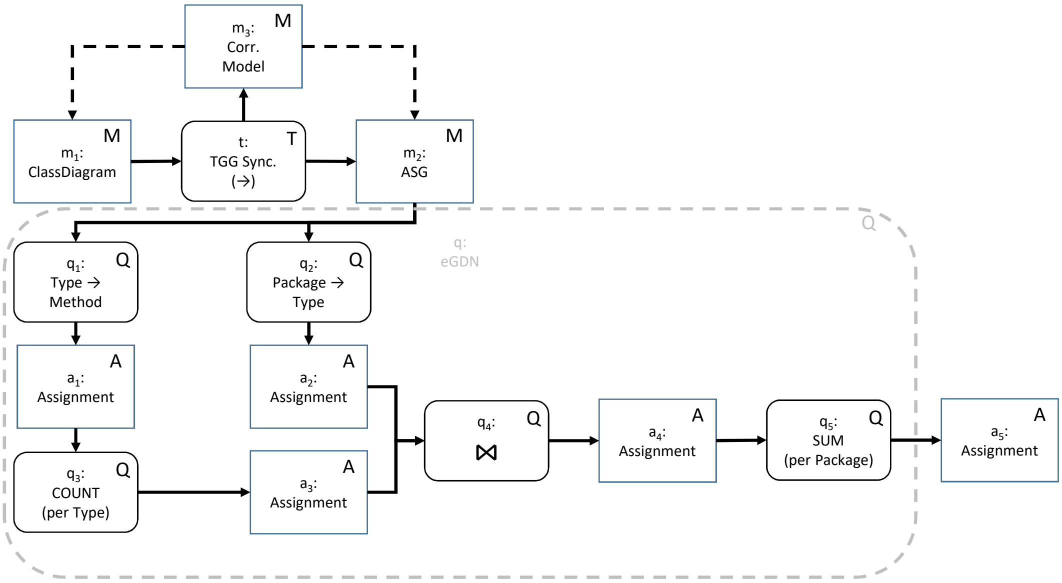

Figure 6.1: Complex example eGDN

Name	**Description**
RETE Nodes	
Node Input	extracts individual nodes of a given type from a model
Edge Input	extracts individual edges of a given type from a model
Join	performs a natural join of assignments stored in two input assignment slots
Anti-Join	performs an anti-join of a left input assignment slot against a right input assignment slot
GDN Nodes	
Pattern Matching	finds matches for a given pattern into a model; supports additional constraints formulated in OCL [58]; supports constraints regarding the existence/absence of matches for other patterns via dependencies to related assignment slots
Property Computation Nodes	
Expression	computes the value of an OCL [58] expression for individual assignments
Group Expression	computes the value of an OCL [58] expression for collections of assignments grouped by certain variables
Group Count	counts the number of assignments in collections of assignments grouped by certain variables
Group Sum	computes the sum of numerical values of a specific variable in collections of assignments grouped by certain variables
Transformation Nodes	
TGG Sync. (→)	performs unidirectional model synchronization of changes from a source to a target and associated correspondence model via a triple graph grammar [63]
TGG Sync. (↔)	performs bidirectional model synchronization of changes between a source, target, and associated correspondence model via a triple graph grammar [63]
Composite Nodes	
eGDN	executes a sub-eGDN to update the contents of exposed slots via Algorithm 1 or Algorithm 3

Table 6.1: Example eGDN node types

Name	**Non-recursive**	**Directions**	**Correct**	**Incremental**
RETE Nodes				
Node Input	✓	all	cond.	✓
Edge Input	✓	all	cond.	✓
Join	✓	all	cond.	✓
Anti-Join	✓	all	cond.	✓
GDN Nodes				
Pattern Matching	✓	all	cond.	(✓)
Property Computation Nodes				
Expression	✓	all	cond.	(✓)
Group Expression	✓	all	cond.	$\sim$
Group Count	✓	all	cond.	✓
Group Sum	✓	all	cond.	✓
Transformation Nodes				
TGG Sync. ($\rightarrow$)	✓	all	cond.	(✓)
TGG Sync. ($\leftrightarrow$)	✓	*	cond.	(✓)
Composite Nodes				
eGDN	?	?	cond.	(✓)

Table 6.2: Properties of example eGDN node types

7 Evaluation

In this chapter, we report on an initial empirical evaluation based on our prototypical implementation. Moreover, we describe how eGDNs can be employed in a typical application scenario, evaluating the developed approach with respect to the requirements from Chapter 3.

7.1 Evaluation of Performance

For an initial empirical evaluation of the proposed approach, we perform an experiment inspired by an application scenario from the software development domain, where an evolving class diagram serves as the basis for generating object-oriented code, which is subsequently analyzed to compute code metrics.

Therefore, we have implemented a simple model transformation from Ecore models [26] to Java abstract syntax graphs [17] via a triple graph grammar. For each class in the class diagram, the transformation creates an interface in the Java abstract syntax graph in a first package, along with an implementation class in a second package. Also, for each attribute of a class in the class diagram, the transformation creates a corresponding field and associated getter and setter methods in the corresponding interface and class in the abstract syntax graph.

In addition, we have realized a model query composed of several subqueries, which counts the number of methods in all types of a Java package. The transformation and query are integrated into an eGDN, which yields the structure displayed in Figure 6.1.

Using our prototypical implementation, which is available under **[30]**, we assign a real-world Ecore model [17] to the class diagram model slot and perform an initial population of the remaining slots via Algorithm 1. To evaluate the scalability of the eGDN, we then apply a number of synthetic updates to the model in the class diagram slot, each of which adds an attribute to each class in the model, and measure the time required for the eGDN to process each such update via Algorithm 1 ("INCREMENTAL"). We compare this to a baseline, where instead, we perform a full recomputation of both the model transformation's and the query's results via non-incremental implementations of the corresponding operations ("BATCH").[1]

---

[1] All experiments were performed on a Linux SMP Debian 4.19.67-2 machine with Intel Xeon E5-2630 CPU (2.3 GHz clock rate) and 386 GB system memory running OpenJDK version 11.0.6. Reported execution time measurements correspond to the mean execution time of 10 runs of the respective experiment.

Figure 7.1 displays the execution times for the first 30 updates. After an initial phase comprising the first 5 updates, where execution time decreases from update to update, the execution time for processing an update to the class diagram via the strategy INCREMENTAL does not change much. In particular, there does not seem to be any trend of increasing execution time related to the growth of the class diagram as additional updates are being performed. In contrast, the execution time of BATCH increases from update to update as the class diagram grows. While it starts out similar to the execution time of INCREMENTAL (larger by factor 1.6), by update 30 the execution time of BATCH has increased to factor 80 compared to the execution time of INCREMENTAL.

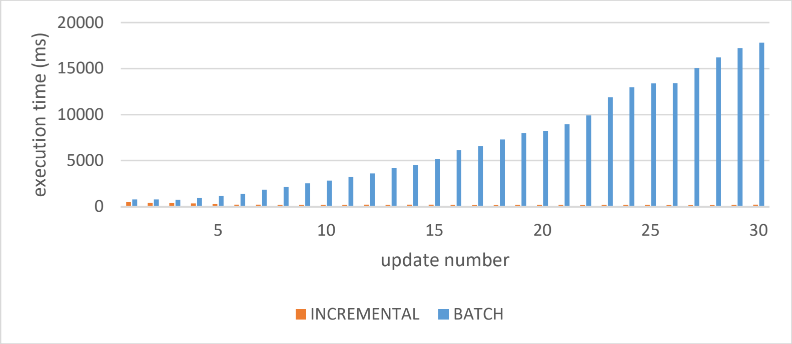

Figure 7.1: Execution time measurements for class diagram updates

The measurements thus indicate that incremental eGDN execution via INCREMENTAL efficiently handles updates to the class diagram, in the sense that execution time only seems to depend on the actual changes rather than the size of the model, indeed affording incrementality. Therefore, eGDNs seem to constitute a suitable formalism for a scalable, modular and incremental realization of networks of model operations for this scenario. The decreasing execution times per update during the initial phase of the experiment can likely be attributed to warming-up effects of the Java virtual machine.

The internal validity of our results is mostly threatened by unexpected behavior of the Java virtual machine, most notably garbage collection. To mitigate such effects, the reported execution time measurements were obtained as the arithmetic mean of multiple runs of the experiment, with the standard deviation of the overall execution time always below 5% of the overall execution time.

The synthetic updates used in the experiment pose a threat to external validity. However, the experiment is inspired by a real-world application scenario and uses a real-world model as its basis and demonstrates the applicability of the eGDN approach in this scenario. The synthetic updates only serve the purpose of allowing a systematic evaluation of our technique's scalability. We hence do not make any quantitative claims regarding our approach in practical application scenarios, but merely consider our experimental results as an indicator for the presented approach's potential. We furthermore do not make claims regarding the generalizability of the approach to other application domains, which would require further evaluation and is left for future work.

7.2 Evaluation of Applicability

In order to investigate the applicability of the developed technique, we consider the following extended example scenario that requires global model management: A class diagram, adhering to a metamodel similar to the one displayed in Figure 2.1, is used to model the structure of a software system under development by means of classes contained in packages. Classes may contain methods, which may in turn reference classes as the method's return type. OCL expressions in a separate model are used to describe the behavior of some of the class diagram's methods. Therefore, the OCL model has its own representation of types corresponding to the classes in the class diagram. This correspondence is captured by means of a linking model, which simply contains dedicated link vertices. A link vertex can either have edges to a class from the class diagram and the corresponding type from the OCL model or edges to a method in the class diagram and the corresponding expression, that is, implementation, in the OCL model.

We consider the following use cases for this setup:

- **Consistency Checking:** The developers want to run automatic and incremental consistency checks that verify that the return type of a method in the class diagram matches the corresponding type of the method's OCL-implementation. The developed consistency check should also work for similar setups that use a different expression language than OCL for the method implementations. An implementation for this use case thus requires a solution satisfying the requirements **R 1.1**, **R 1.2**, **R 2.2**, and **R 2.3**.

- **Code Generation:** The developers want to automatically and incrementally generate Java code in the form of an ASG from the class diagram. In addition, Java implementations for the class diagram's methods should be generated from the methods' OCL implementations. In the end, the resulting Java code fragments for the two models should be integrated and analyzed for some code metrics. An implementation for this use case thus requires a solution satisfying the requirements **R 2.1**, **R 2.2**, and **R 2.3**.

- **Megamodel Reuse:** After developing the automatic consistency checking and code generation, the developers want to reuse the same two operations in another project with a similar set of models. An implementation for this use case thus requires a solution satisfying the requirements **R 3.1.2**, **R 3.2**, **R 3.3**, and **R 3.4**.

In order to allow global model management for all three use cases, a solution also has to enable the modeling of a network of different kinds of model operations over a set of potentially integrated models, that is, a solution has to satisfy requirement **R 3.1.1**.

Figures 7.3, 7.2, and 7.4 visualize example eGDN implementations for the *Consistency Checking*, *Code Generation*, and *Megamodel Reuse* use cases, respectively.

As displayed in Figure 7.3, the *Consistency Checking* use case is realized via four Pattern Matching query nodes that extract certain simple patterns from the base models and make them accessible in a generalized format via assignment slots. Then, a complex query operation, which is composed of three Join query operations and an Anti-Join query operation (labelled $\triangleright$), realizes the actual consistency check by finding all the combinations of a method from the class diagram, its implementation from the OCL model, and the associated return class respectively expression type, where the return class and expression type do not correspond. Thus, the eGDN-based approach in this case fulfills the requirements **R 1.1** and **R 2.2**, as it implements a consistency check over a set of models integrated via integration links. Furthermore, the resulting implementation is reusable for different modeling languages that offer similar functionality via the generic interface provided by the assignment slots a_1, a_2, a_3, and a_7, satisfying requirement **R 1.2**. The example eGDN also demonstrates how more complex model operations can be composed from simpler operations, satisfying requirement **R 2.1**, and provides an incremental execution scheme for these operations, satisfying requirement **R 2.3**. In particular, via Algorithm 2, it can be verified that Algorithm 1 provides a means of executing the eGDN that guarantees both correct results and termination for changes to any combination of the three base models.

The eGDN shown in Figure 7.2 realizes the *Code Generation* use case via a combination of two TGG Synchronizations that translate the class diagram and OCL model into Java ASGs. The two Java models are integrated via a dedicated linking model, which is produced by a unidirectional model transformation from the original linking model and the correspondence models created by the TGG Synchronizations. Finally, query operations can be executed over the ASGs to compute code metrics. Using Algorithm 2 to analyze the eGDN, it can be determined that terminating execution via Algorithm 1 can be guaranteed for changes to any combination of the three base models. This example shows how the eGDN provides a unified, modular notion of model operations along with an incremental execution scheme and demonstrates the composition of model operations, satisfying requirements **R 2.1**, **R 2.2**, and **R 2.3**.

Together, the eGDNs in Figure 7.3 and 7.2 also illustrate how eGDNs can be used as a megamodeling language, supporting different kinds of model operations, including model properties (like the metrics computed in the Code Generation use case), model consistency (like the consistency condition in the Consistency Checking use case), and model transformation and synchronization (like the transformation and synchronizations in the Code Generation use case). It also shows how integration views (like the cross-model consistency query results in slot a_8 in Figure 7.3) and traceability links (like the correspondence models produced by the TGG Synchronizations) can be represented in the language. While not present in the example eGDNs, the class diagram and OCL metamodel are models themselves and could simply be made explicit by including them in dedicated model slots. Well-formedness conditions for metamodels or regular models can be realized and treated as regular query operations. eGDNs thus satisfy the requirement **R 3.1.1**.

Finally, the eGDN realization of the *Megamodel Reuse* use case in Figure 7.4 considers the eGDNs from Figure 7.3 and 7.2 as operation nodes in an overarching eGDN. This exemplifies how eGDNs offer modularity and incrementality at the megamodel level by considering sub-eGDNs as regular operations that can be executed via the general execution scheme, which also permits the accumulation of several changes before execution. The example thus illustrates the satisfaction of requirements **R 3.1.2**, **R 3.2**, and **R 3.4**. The eGDN also demonstrates how slots act as interfaces for these megamodel operations, satisfying requirement **R 3.3**.

Thus, eGDNs can be employed to realize the functionality required by the three example use cases, satisfying the requirements regarding model operations, modeling languages integration, and megamodels introduced in Section 3 in this scenario.

Table 7.1 summarizes the coverage of the requirements by the example use cases and the eGDN approach, with "○" denoting that the realization of a use case relates to a requirement and "✓" indicating that a requirement is satisfied by eGDNs in this scenario.

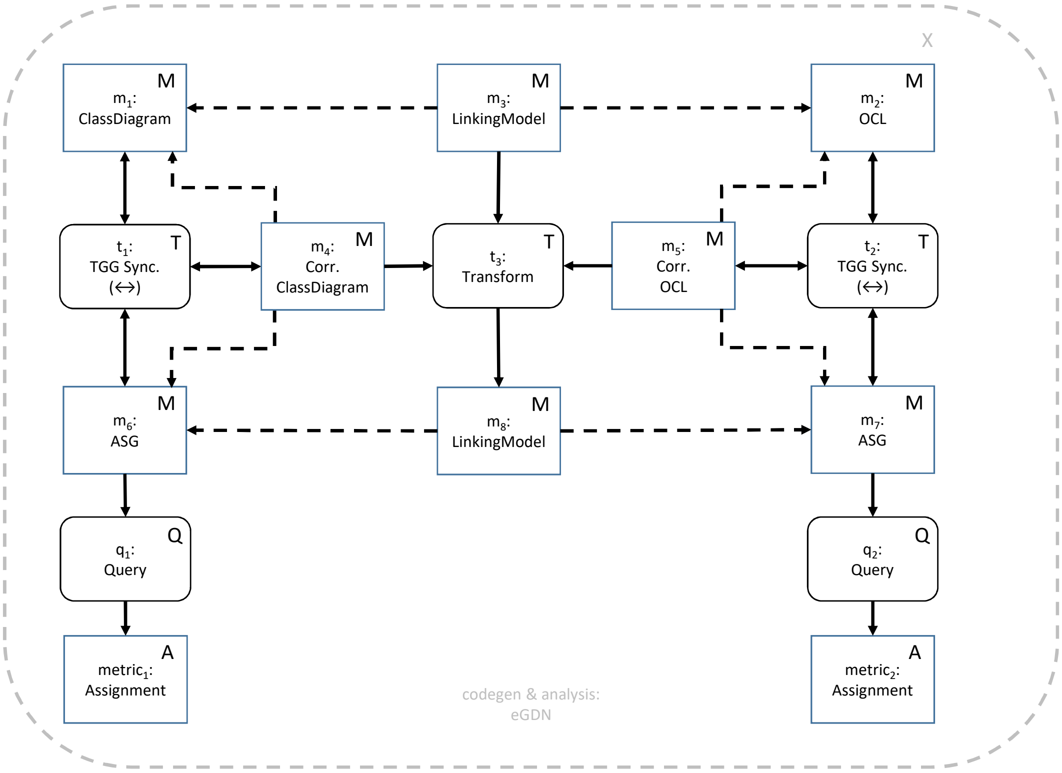

Figure 7.2: Sample eGDN realizing the Code Generation use case

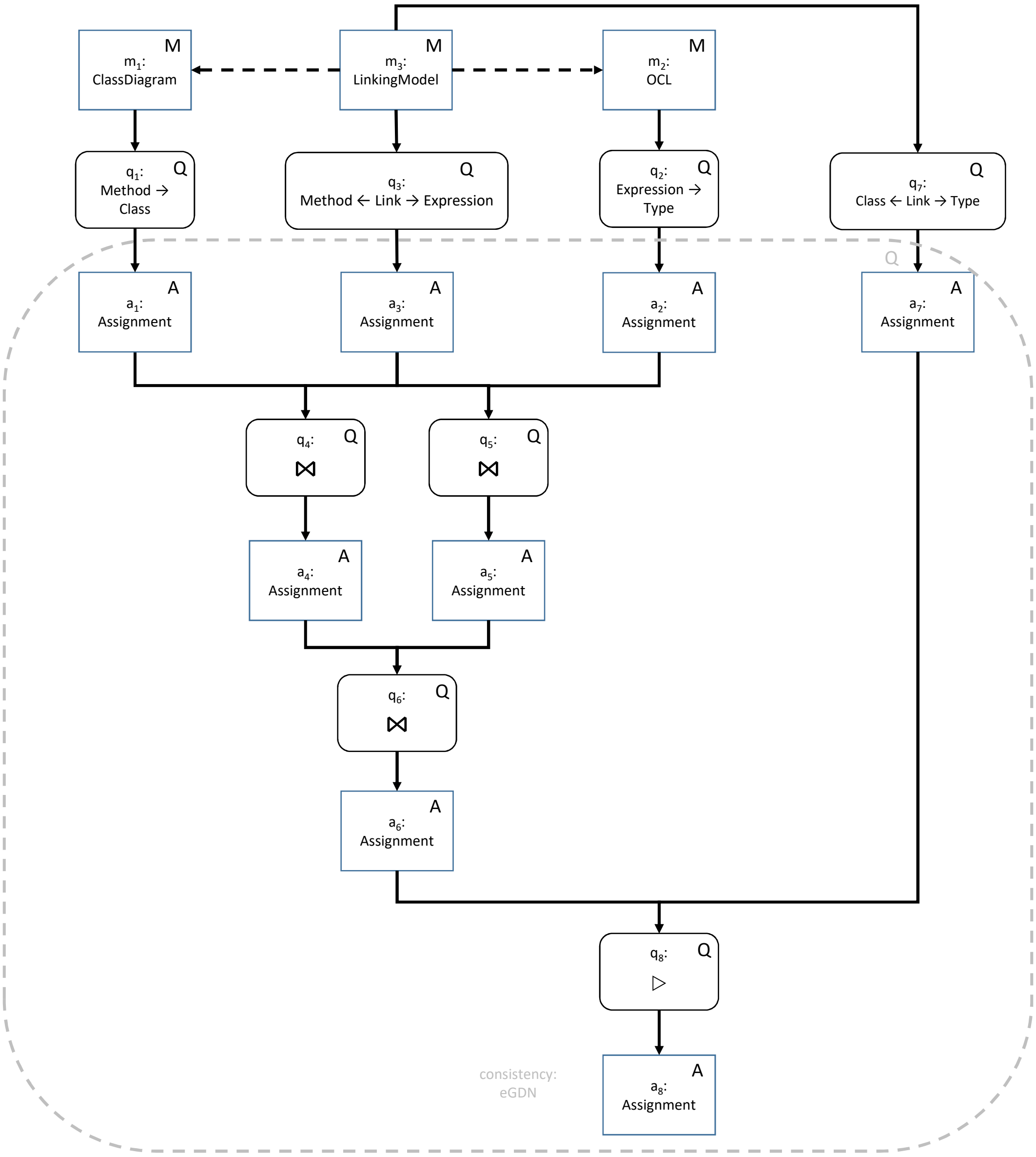

Figure 7.3: Sample eGDN realizing the Consistency Checking use case

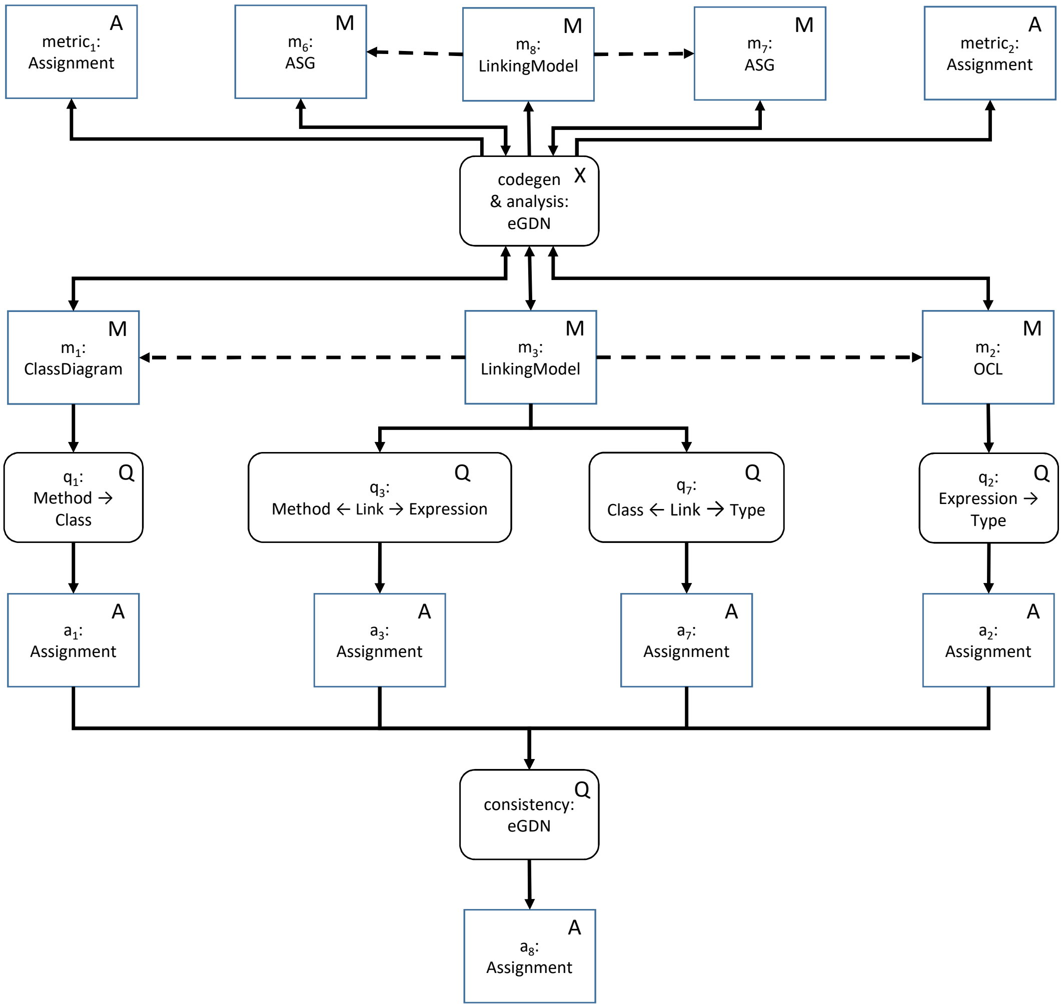

Figure 7.4: Sample eGDN realizing the Megamodel Reuse use case

	Consistency Checking	Code Generation	Megamodel Reuse	eGDNs
R 1.1: modeling languages integration	○			✓
R 1.2: interfaces for embedding of modeling languages	○			✓
R 2.1: composition of model operations		○		✓
R 2.2: model operations over integrated models	○	○		✓
R 2.3: execution scheme for model operations	○	○		✓
R 3.1.1: megamodeling language	○	○	○	✓
R 3.1.2: megamodel operation module concept			○	✓
R 3.2: robust megamodel execution scheme			○	✓
R 3.3: megamodel interfaces			○	✓
R 3.4: asynchronous megamodel execution scheme			○	✓

Table 7.1: Coverage of requirements from Chapter 3

8 Conclusion

In this report, we have developed a further generalization of the GDN mechanism called eGDNs, which enables the modular and incremental construction and execution of complex networks of model operations, including model properties, model consistency, model transformation and model synchronization. In addition to a formal definition of eGDNs, we have provided incremental algorithms for their execution. Moreover, we have presented a number of example eGDN nodes that we have prototypically implemented in order to perform an initial empirical evaluation of the approach regarding scalability. Our experiments, which are based on an application scenario from the software development domain, indicate that the introduced technique can be employed to realize efficient Global Model Management. Moreover, we have conceptually evaluated our approach against identified requirements of global model management solutions.

In future work, we plan to perform a more extensive evaluation with respect to both expressiveness and performance of eGDNs in real application scenarios. This may also involve the implementation of additional types of eGDN nodes and may ultimately result in the implementation of true tool support for the specification and execution of eGDNs. Furthermore, we will investigate how the presented concepts can be extended to the case of evolving modeling landscapes that consist of multiple distinct versions. We will also explore how such an extension may help alleviate problems such as potential infinite loops or overwriting of user edits in eGDN execution via the derivation of additional versions.

Acknowledgements

This work was developed mainly in the course of the project modular and incremental Global Model Management (project number 336677879), which is funded by the Deutsche Forschungsgemeinschaft.

Bibliography

[1] *AM3 Project Homepage*. `https://wiki.eclipse.org/AM3`. Last accessed 25 January 2023.

[2] *AMW Project Homepage*. `https://projects.eclipse.org/projects/modeling.gmt.amw/`. Last accessed 25 January 2023.

[3] *ATL Project Homepage*. `https://eclipse.org/atl/`. Last accessed 25 January 2023.

[4] *ATLFlow Project Homepage*. `http://opensource.urszeidler.de/ATLflow/`. Last accessed 25 January 2023.

[5] *AToMPM Project Homepage*. `https://atompm.github.io/`. Last accessed 25 January 2023.

[6] T. Beyhl, D. Blouin, H. Giese, and L. Lambers. "On the operationalization of graph queries with generalized discrimination networks". In: *International Conference on Graph Transformation*. Springer. 2016, pages 170–186.

[7] T. Beyhl, R. Hebig, and H. Giese. "A Model Management Framework for Maintaining Traceability Links". In: *Software Engineering 2013 Workshopband*. Edited by S. Wagner and H. Lichter. Volume P-215. Lecture Notes in Informatics (LNI). Aachen: Gesellschaft für Informatik (GI), Feb. 2013, pages 453–457.

[8] J. Bézivin, F. Jouault, P. Rosenthal, and P. Valduriez. "Modeling in the Large and Modeling in the Small". In: *Model Driven Architecture*. Volume 3599/2005. Lecture Notes in Computer Science (LNCS). Springer-Verlag, 2005, pages 33–46.

[9] J. Bézivin, F. Jouault, P. Rosenthal, and P. Valduriez. "Modeling in the large and modeling in the small". In: *Model Driven Architecture*. Springer, 2004, pages 33–46.

[10] J. Bézivin, F. Jouault, and P. Valduriez. "On the Need for Megamodels". In: *Proceedings of the OOPSLA/GPCE: Best Practices for Model-Driven Software Development workshop, 19th Annual ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications*. 2004.

[11] X. Blanc, A. Mougenot, I. Mounier, and T. Mens. "Incremental Detection of Model Inconsistencies Based on Model Operations". In: *CAiSE '09: Proceedings of the 21st International Conference on Advanced Information Systems Engineering, Amsterdam, The Netherlands*. Volume 5565/2009. Berlin, Heidelberg: Springer Verlag, Aug. 2009, pages 32–46.

[12] A. Blouin, B. Combemale, B. Baudry, and O. Beaudoux. “Kompren: modeling and generating model slicers”. In: *Software & Systems Modeling* 14.1 (2015), pages 321–337.

[13] D. Blouin, Y. Eustache, and J.-P. Diguet. “Extensible Global Model Management with Meta-model Subsets and Model Synchronization”. In: *Proceedings of the 2nd International Workshop on The Globalization of Modeling Languages co-located with ACM/IEEE 17th International Conference on Model Driven Engineering Languages and Systems, GEMOC@Models 2014, Valencia, -*. 2014, pages 43–52.

[14] D. Blouin, G. Ochoa Ruiz, Y. Eustache, and J.-P. Diguet. “Kaolin: a System-level AADL Tool for FPGA Design Reuse, Upgrade and Migration”. In: *NASA/ESA International Conference on Adaptive Hardware and Systems (AHS)*. Montréal, Canada, June 2015.

[15] M. Boddy, M. Michalowski, A. Schwerdfeger, H. Shackleton, S. Vestal, and A. Enterprises. “FUSED: A Tool Integration Framework for Collaborative System Engineering”. In: *Analytic Virtual Integration of Cyber-Physical Systems Workshop*. 2011.

[16] H. Bruneliere, E. Burger, J. Cabot, and M. Wimmer. “A feature-based survey of model view approaches”. In: *Software & Systems Modeling* 18.3 (2019), pages 1931–1952.

[17] H. Bruneliere, J. Cabot, F. Jouault, and F. Madiot. “MoDisco: a generic and extensible framework for model driven reverse engineering”. In: *Proceedings of the IEEE/ACM international conference on Automated software engineering*. 2010. DOI: `10.1145/1858996.1859032`.

[18] H. Bruneliere, J. G. Perez, M. Wimmer, and J. Cabot. “EMF Views: A View Mechanism for Integrating Heterogeneous Models”. In: *Conceptual Modeling - 34th International Conference, ER 2015, Stockholm Sweden, October 19-22, 2015, Proceedings*. 2015, pages 317–325.

[19] J. Cabot and E. Teniente. “Incremental Evaluation of OCL Constraints”. In: *CAiSE’06: 18th International Conference on Advanced Information Systems Engineering, Luxembourg, Luxembourg*. Volume 4001/2006. Lecture Notes in Computer Science (LNCS). Springer Verlag, May 2006, pages 81–95.

[20] *CoEST Project Homepage*. `http://www.coest.org/`. Last accessed 25 January 2023.

[21] *Composite EMF Models Project Homepage*. `http://www.mathematik.uni-marburg.de/~swt/compoemf/`. Last accessed 25 January 2023.

[22] C. Debreceni, A. Horvath, A. Hegedus, Z. Ujhelyi, I. Rath, and D. Varro. “Query-driven Incremental Synchronization of View Models”. In: *Proceedings of the 2Nd Workshop on View-Based, Aspect-Oriented and Orthographic Software Modelling*. VAO ’14. New York, NY, USA: ACM, 2014, 31:31–31:38.

[23] A. Egyed. “Instant Consistency Checking for the UML”. In: *ICSE ’06: Proceedings of the 28th International Conference on Software Engineering*. Shanghai, China, 20-28 May 2006, pages 381–390.

[24] H. Ehrig, K. Ehrig, U. Prange, and G. Taentzer. *Fundamentals of algebraic graph transformation*. EATCS. Springer, 2006. DOI: `10.1007/3-540-31188-2`.

[25] *EMF-IncQuery Project Homepage*. `https://www.eclipse.org/incquery/`. Last accessed 25 January 2023.

[26] *EMF: Eclipse Modeling Framework*. `https://www.eclipse.org/modeling/emf/`. Last accessed 25 November 2022.

[27] *EMF Views Project Homepage*. `http://atlanmod.github.io/emfviews/`. Last accessed 25 January 2023.

[28] *Epsilon Project Homepage*. `http://eclipse.org/epsilon/`. Last accessed 25 January 2023.

[29] A. Etien, A. Muller, T. Legrand, and X. Blanc. “Combining Independent Model Transformations”. In: *Proceedings of the 2010 ACM Symposium on Applied Computing*. SAC ’10. New York, NY, USA: ACM, 2010, pages 2237–2243.

[30] *Extended Generalized Discrimination Networks Evaluation Artifacts*. `https://github.com/hpi-sam/Extended-GDNs`. Last accessed 25 November 2022.

[31] J.-M. Favre. “Foundations of Model (Driven) (Reverse) Engineering – Episode I: Story of The Fidus Papyrus and the Solarus”. In: *Post-Proceedings of Dagstuhl Seminar on Model Driven Reverse Engineering*. 2004.

[32] J.-M. Favre, R. Lämmel, and A. Varanovich. “Modeling the linguistic architecture of software products”. In: *Proceedings of the 15th international conference on Model Driven Engineering Languages and Systems*. MODELS’12. Berlin, Heidelberg: Springer-Verlag, 2012, pages 151–167.

[33] F. Fleurey, B. Baudry, R. France, and S. Ghosh. “Models in Software Engineering”. In: edited by H. Giese. Berlin, Heidelberg: Springer-Verlag, 2008. Chapter A Generic Approach for Automatic Model Composition, pages 7–15.

[34] C. L. Forgy. “Rete: A fast algorithm for the many pattern/many object pattern match problem”. In: *Readings in Artificial Intelligence and Databases*. Elsevier, 1989, pages 547–559.

[35] *FUSED Project Homepage*. `http://www.adventiumlabs.com/`. Last accessed 25 January 2023.

[36] E. Gamma, R. Helm, R. Johnson, R. E. Johnson, J. Vlissides, et al. *Design patterns: elements of reusable object-oriented software*. Pearson Deutschland GmbH, 1995.

[37] H. Giese, S. Neumann, and S. Hildebrandt. "Model Synchronization at Work: Keeping SysML and AUTOSAR Models Consistent". In: *Graph Transformations and Model Driven Engineering - Essays Dedicated to Manfred Nagl on the Occasion of his 65th Birthday*. Edited by G. Engels, C. Lewerentz, W. Schäfer, A. Schürr, and B. Westfechtel. Volume 5765. Lecture Notes in Computer Science. Springer Berlin / Heidelberg, 2010, pages 555–579.

[38] H. Giese, S. Neumann, O. Niggemann, and B. Schätz. "Model-Based Integration". In: *Model-Based Engineering of Embedded Real-Time Systems - International Dagstuhl Workshop, Dagstuhl Castle, Germany, November 4-9, 2007. Revised Selected Papers*. Edited by H. Giese, G. Karsai, E. Lee, B. Rumpe, and B. Schätz. Volume 6100. Lecture Notes in Computer Science. Springer, 2011, pages 17–54.

[39] H. Giese and R. Wagner. "From model transformation to incremental bidirectional model synchronization". In: *Software & Systems Modeling* 8.1 (2009), pages 21–43.

[40] J. Gleitze, H. Klare, and E. Burger. "Finding a universal execution strategy for model transformation networks". In: *International Conference on Fundamental Approaches to Software Engineering*. Springer, Cham. 2021, pages 87–107.

[41] I. Groher, A. Reder, and A. Egyed. "Incremental Consistency Checking of Dynamic Constraints". In: *Fundamental Approaches to Software Engineering*. Edited by D. Rosenblum and G. Taentzer. Volume 6013. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2010, pages 203–217.

[42] A. Habel and K.-H. Pennemann. "Correctness of high-level transformation systems relative to nested conditions". In: *Mathematical Structures in Computer Science* 19.2 (2009), pages 245–296.

[43] E. N. Hanson, S. Bodagala, and U. Chadaga. "Trigger condition testing and view maintenance using optimized discrimination networks". In: *IEEE Transactions on Knowledge and Data Engineering* 14.2 (2002), pages 261–280.

[44] R. Hebig, A. Seibel, and H. Giese. "On the Unification of Megamodels". In: *Proceedings of the 4th International Workshop on Multi-Paradigm Modeling (MPM 2010)*. Edited by V. Amaral, H. Vangheluwe, C. Hardebolle, L. Lengyel, T. Magaria, J. Padberg, and G. Taentzer. Volume 42. Electronic Communications of the EASST. 2011.

[45] R. Heckel, J. M. Küster, and G. Taentzer. "Confluence of typed attributed graph transformation systems". In: *International Conference on Graph Transformation*. Springer. 2002, pages 161–176.

[46] S. J. Herzig, A. Qamar, and C. J. Paredis. "An Approach to Identifying Inconsistencies in Model-based Systems Engineering". In: *Procedia Computer Science* 28.0 (2014). 2014 Conference on Systems Engineering Research, pages 354–362. ISSN: 1877-0509. DOI: `http://dx.doi.org/10.1016/j.procs.2014.03.044`.

[47] S. J. Herzig and C. J. Paredis. "Bayesian Reasoning Over Models". In: *11th Workshop on Model Driven Engineering, Verification and Validation MoDeVVa 2014*. 2014, page 69.

[48] A. Hessellund and A. Wasowski. "Interfaces and Metainterfaces for Models and Metamodels". In: *Model Driven Engineering Languages and Systems*. Edited by K. Czarnecki, I. Ober, J.-M. Bruel, A. Uhl, and M. Wolter. Volume 5301. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2008, pages 401–415.

[49] J. Johannes and U. Assmann. "Concern-Based (de)composition of Model-Driven Software Development Processes". In: *Model Driven Engineering Languages and Systems*. Edited by D. Petriu, N. Rouquette, and O. Haugen. Volume 6395. Lecture Notes in Computer Science. 10.1007/978-3-642-16129-2_5. Springer Berlin / Heidelberg, 2010, pages 47–62.

[50] S. Jurack and G. Taentzer. "A Component Concept for Typed Graphs with Inheritance and Containment Structures". In: *Graph Transformations - 5th International Conference, ICGT 2010 Enschede, The Netherlands, September 27 - October 2, 2010. Proceedings*. 2010, pages 187–202.

[51] H. Klare, M. E. Kramer, M. Langhammer, D. Werle, E. Burger, and R. Reussner. "Enabling consistency in view-based system development—the vitruvius approach". In: *Journal of Systems and Software* 171 (2021), page 110815.

[52] *Kompose Project Homepage*. `http://www.kermeta.org/mdk/kompose`. Last accessed 25 January 2023.

[53] H. König and Z. Diskin. "Efficient consistency checking of interrelated models". In: *European Conference on Modelling Foundations and Applications*. Springer. 2017, pages 161–178.

[54] D. Langsweirdt, N. Boucke, and Y. Berbers. "Architecture-Driven Development of Embedded Systems with ACOL". In: *Object/Component/Service-Oriented Real-Time Distributed Computing Workshops (ISORCW), 2010 13th IEEE International Symposium on*. May 2010, pages 138–144.

[55] H. Lochmann and A. Hessellund. "An Integrated View on Modeling with Multiple Domain-Specific Languages". In: *Proceedings of the IASTED International Conference Software Engineering (SE 2009)*. ACTA Press, Feb. 2009, pages 1–10.

[56] L. Lúcio, S. Mustafiz, J. Denil, H. Vangheluwe, and M. Jukss. "FTG+PM: An Integrated Framework for Investigating Model Transformation Chains". In: *SDL 2013: Model-Driven Dependability Engineering*. Edited by F. Khendek, M. Toeroe, A. Gherbi, and R. Reed. Volume 7916. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2013, pages 182–202.

[57] *MoTE Project Homepage*. `http://www.mdelab.org/mdelab-projects/mote-a-tgg-based-model-transformation-engine/`.

[58] Object Management Group. *Object Constraint Language Specification*. `https://www.omg.org/spec/OCL/`. Last accessed 25 November 2022.

[59] *OSLC Project Homepage*. `http://open-services.net/`. Last accessed 25 January 2023.

[60] R. F. Paige, D. S. Kolovos, L. Rose, N. Drivalos, and F. A. C. Polack. "The Design of a Conceptual Framework and Technical Infrastructure for Model Management Language Engineering". In: *Engineering of Complex Computer Systems, 2009 14th IEEE International Conference on*. June 2009, pages 162–171.

[61] C. C. Raţiu, W. K. Assunção, R. Haas, and A. Egyed. "Reactive links across multi-domain engineering models". In: *Proceedings of the 25th International Conference on Model Driven Engineering Languages and Systems*. 2022, pages 76–86.

[62] J. E. Rivera, D. Ruiz-Gonzalez, F. Lopez-Romero, J. Bautista, and A. Vallecillo. "Orchestrating ATL Model Transformations". In: *In Proc. of MtATL 2009: 1st International Workshop on Model Transformation with ATL*. Edited by F. Jouault. Nantes, France, July 2009, pages 34–46.

[63] A. Schürr. "Specification of graph translators with triple graph grammars". In: *Proc. of the* 20^t*h International Workshop on Graph-Theoretic Concepts in Computer Science*. Edited by E. W. Mayr, G. Schmidt, and G. Tinhofer. Volume 903. Lecture Notes in Computer Science. Herrsching, Germany: Spinger Verlag, June 1994, pages 151–163.

[64] A. Seibel, R. Hebig, and H. Giese. "Traceability in Model-Driven Engineering: Efficient and Scalable Traceability Maintenance". In: *Software and Systems Traceability*. Edited by J. Cleland-Huang, O. Gotel, and A. Zisman. Springer London, 2012, pages 215–240.

[65] A. Seibel, R. Hebig, S. Neumann, and H. Giese. "A Dedicated Language for Context Composition and Execution of True Black-Box Model Transformations". In: *4th International Conference on Software Language Engineering (SLE 2011) , Braga, Portugal*. July 2011.

[66] A. Seibel, S. Neumann, and H. Giese. "Dynamic hierarchical mega models: comprehensive traceability and its efficient maintenance". In: *Software & Systems Modeling* 9.4 (2010), pages 493–528.

[67] A. Seibel, S. Neumann, and H. Giese. "Dynamic hierarchical mega models: comprehensive traceability and its efficient maintenance". In: *Software and Systems Modeling* 9.4 (2010), pages 493–528.

[68] G. Simko, T. Levendovszky, S. Neema, E. Jackson, T. Bapty, J. Porter, and J. Sztipanovits. "Foundation for model integration: Semantic backplane". In: *ASME 2012 International Design Engineering Technical Conferences and Computers and Information in Engineering Conference*. American Society of Mechanical Engineers. 2012, pages 1077–1086.

[69] P. Stevens. "Connecting software build with maintaining consistency between models: towards sound, optimal, and flexible building from megamodels". In: *Software and Systems Modeling* 19.4 (2020), pages 935–958.

[70] P. Stevens. "Maintaining consistency in networks of models: bidirectional transformations in the large". In: *Software and Systems Modeling* 19.1 (2020), pages 39–65.

[71] Z. Ujhelyi, G. Bergmann, A. Hegedus, A. Horvath, B. Izso, I. Rath, Z. Szatmari, and D. Varro. "EMF-IncQuery: An integrated development environment for live model queries". In: *Science of Computer Programming* 98, Part 1 (2015). Fifth issue of Experimental Software and Toolkits (EST): A special issue on Academics Modelling with Eclipse (ACME2012), pages 80–99.

[72] H. Vangheluwe, J. de Lara, and P. J. Mosterman. "An introduction to multi-paradigm modelling and simulation". In: *Proceedings of the AIS2002 Conference (2002)*. 2002.

[73] D. Varró, G. Bergmann, Á. Hegedüs, Á. Horváth, I. Ráth, and Z. Ujhelyi. "Road to a reactive and incremental model transformation platform: three generations of the VIATRA framework". In: *Software & Systems Modeling* 15.3 (2016), pages 609–629.

[74] A. Vignaga, F. Jouault, M. Bastarrica, and H. Brunelière. "Typing artifacts in megamodeling". In: *Software & Systems Modeling* 12 (2013), pages 105–119.

Current Technical Reports of the Hasso-Plattner-Institut

Vol.	ISBN	Title	Authors/Editors
153	978-3-86956-551-4	**Human pose estimation for decubitus prophylaxis**	Benedikt Weber
152	978-3-86956-550-7	**Abschlussbericht Forschungsprojekt „RailChain“**	Lukas Pirl
151	978-3-86956-547-7	**HPI Future SOC Lab – Proceedings 2018**	Christoph Meinel, Andreas Polze, Karsten Beins, Rolf Strotmann, Ulrich Seibold, Kurt Rödszus, Jürgen Müller
150	978-3-86956-546-0	**openHPI : 10 Jahre MOOCs am Hasso-Plattner-Institut**	Christoph Meinel, Christian Willems, Thomas Staubitz, Dominic Sauer, Christiane Hagedorn
149	978-3-86956-545-3	**Implementing a crowd-sourced picture archive for Bad Harzburg**	Rieke Freund, Jan Philip Rätsch, Franziska Hradilak, Benedikt Vidic, Oliver Heß, Nils Lißner, Hendrik Wölert, Jens Lincke, Tom Beckmann, Hirschfeld Robert
148	978-3-86956-544-6	**openHPI : 10 Years of MOOCs at the Hasso Plattner Institute**	Christoph Meinel, Christian Willems, Thomas Staubitz, Dominic Sauer, Christiane Hagedorn
147	978-3-86956-533-0	**Modeling and formal analysis of meta-ecosystems with dynamic structure using graph transformation**	Boris Flotterer, Maria Maximova, Sven Schneider, Johannes Dyck, Christian Zöllner, Holger Giese, Christelle Hély, Cédric Gaucherel
146	978-3-86956-532-3	**Probabilistic metric temporal graph logic**	Sven Schneider, Maria Maximova, Holger Giese
145	978-3-86956-528-6	**Learning from failure : a history-based, lightweight test prioritization technique connecting software changes to test failures**	Falco Dürsch, Patrick Rein, Toni Mattis, Robert Hirschfeld

ISBN 978-3-86956-555-2
ISSN 1613-5652