Integration of Triple Graph Grammars in Vitruvius

Master's Thesis of

Robin Schulz

At the KIT Department of Informatics
KASTEL – Institute of Information Security and Dependability

First examiner:	Prof. Dr. Ralf Reussner
Second examiner:	Prof. Dr.-Ing. Anne Koziolek
First advisor:	Lars König, M.Sc.
Second advisor:	Thomas Weber, M.Sc.

18 November 2024 – 18 May 2025

Karlsruher Institut für Technologie
Fakultät für Informatik
Postfach 6980
76128 Karlsruhe

I declare that I have not used any generative AI tools to develop and write the enclosed thesis besides for spell checking and grammar correction.

Integration of Triple Graph Grammars in Vitruvius (Master's Thesis)

I declare that I have developed and written the enclosed thesis completely by myself. I have not used any other than the aids that I have mentioned. I have marked all parts of the thesis that I have included from referenced literature, either in their original wording or paraphrasing their contents. I have followed the by-laws to implement scientific integrity at KIT.

Karlsruhe, 18 May 2025

[illegible]
. . . .
(Robin Schulz)

Abstract

As complex software systems or software-intensive systems often are described by multiple artifacts written in different languages, information redundancy and dependencies can be present between these artifacts, which can also be viewed as models. This raises the question of consistency between those artifacts. If consistency-relevant changes are made to one model, some effort, in the form of adapting relevant information in the other models, needs to be made to restore consistency within the whole system. As an approach to reduce the manual share of that effort, the concept of *Virtual Single Underlying Models* (V-SUMs) was proposed. There, models are modified only via *views* and *consistency preservation rules* (CPRs) are defined to specify how consistency is preserved if changes are made to one model. These CPRs can be defined in specialized languages, and in this work they are supplemented by another transformation language, using an existing and mature general-purpose approach called *Triple Graph Grammars* (TGGs) that allows for expressing complex relations by specifying graphical patterns called *TGG rules*. To be able to stay close to the paradigm of delta-based consistency preservation, the application of TGG rules as CPRs has to be based on information about how and what changes were applied, i.e., a sequence of changes. A concept is presented that allows matching TGG rules to sequences of changes to a model to be able to apply the rule to the target model and thus apply the CPR represented by the TGG rule. This concept is prototypically implemented using the Vitruvius framework for V-SUMs and *eMoflon::IBeX*, which is a TGG framework. The concept is evaluated with respect to correctness, consistency completeness, and performance. While revealing drawbacks introduced by *eMoflon::IBeX*, the results indicate no limitations to correctness and only minor limitations to consistency completeness. In comparison to *HiPE* [17], an approach that ignores the change sequence information, the prototype performs well in small to medium (128 changes in the sequence) problem sizes, but results indicate a worse runtime complexity than *HiPE*. Propositions to mitigate the scalability issues with larger change sequences are presented.

Zusammenfassung

Da komplexe Software-Systeme oder Software-intensive Systeme häufig aus verschiedenen Artefakten bestehen, die in verschiedenen Sprachen beschrieben sind, treten Informations-Redundanzen und informationelle Abhängigkeiten zwischen diesen Abhängigkeiten auf, was die Frage nach Konsistenz zwischen diesen Artefakten, welche auch als Modelle angesehen werden können, aufwirft. Wenn konsistenzrelevante Änderungen an einem Modell vorgenommen werden, muss Aufwand in Form von Anpassungen von Informationen in anderen Modellen betrieben werden, um die Konsistenz im Gesamtsystem wiederherzustellen. Als Ansatz zur möglichst weitestgehenden Reduktion des manuellen Anteils an diesem Aufwand wurde das Konzept von *Virtual Single Underlying Models* (V-SUMs) vorgestellt, in welchem Modelle ausschließlich mittels *Sichten* modifiziert werden können und *consistency preservation rules* (CPRs) definiert werden, die spezifizieren, wie die Konsistenz im Gesamtsystem erhalten wird, wenn Änderungen an einem Modell vorgenommen werden. Diese CPRs können in auf diesen Anwendungsfall spezialisierten Sprachen beschrieben werden, und in dieser Arbeit werden sie durch einen Ansatz ergänzt, welcher einen existierenden allgemeineren Ansatz namens *Triple Graph Grammars* (TGGs) nutzt, der es ermöglicht, komplexe Beziehungen zwischen Modellen darzustellen, indem Graph-Muster spezifiziert werden, die TGG-Regeln genannt werden. Um dem Paradigma der Delta-basierten Konsistenz-Erhaltung gerecht werden zu können, muss die Anwendung der TGG-Regeln als CPRs auf Informationen darüber basieren, welche Änderungen und in welcher Reihenfolge diese angewendet wurden – es wird also eine Änderungs-Sequenz benötigt. In dieser Arbeit wird ein Konzept präsentiert, welches es ermöglicht, TGG-Regeln mit Änderungs-Sequenzen bezüglich eines Modells abzugleichen, um dann die Regeln zu nutzen, um die Änderung in ein anderes Modell zu übertragen, und damit die durch die TGG-Regel dargestellte CPR anzuwenden. Das Konzept wird prototypisch mit dem Vitruvius-Rahmenwerk für V-SUMs und *eMoflon::IBeX*, einem TGG-Rahmenwerk, implementiert. Dieser Prototyp wird genutzt, um das Konzept hinsichtlich Korrektheit, Konsistenz-Vollständigkeit und Performanz zu evaluieren. Die Ergebnisse zeigen zwar Einschränkungen auf, die durch *eMoflon::IBeX* entstehen, aber bezüglich des Konzepts können keine Einschränkungen hinsichtlich der Korrektheit und nur kleinere Einschränkungen hinsichtlich der Konsistenz-Vollständigkeit festgestellt werden. Im Vergleich zu *HiPE* [17], einem bestehenden Ansatz, welcher die Änderungssequenz-Information ignoriert, ist die Laufzeit des Prototyps für kleinen bis mittleren Problemgrößen (128 Änderungen in der Änderungssequenz) vergleichbar oder besser. Für größere Sequenzen divergieren die Laufzeiten und der Ansatz zeigt eine schlechtere Laufzeit-Komplexität als *HiPE* auf. Zur Abmilderung oder Lösung dieser Thematik werden in dieser Arbeit Vorschläge präsentiert und diskutiert.

Contents

List of Figures

List of Tables

1. Introduction

The development of many large modern IT systems and cyber-physical systems (CPS) is characterized by the use of different languages and tools to describe different parts and aspects of the system under development, e.g., design, implementation, and documentation and different subdomains and subsystems. The use of different languages produces different informational artifacts describing the system and dependencies between those artifacts.

As an approach to cope with increasingly complex conglomerates of different artifacts that all describe a single system, *Model Driven Engineering* (MDE) [24] has become a strategy with industrial relevance [19, 20, 39]. In the software context, different groups of developers of such systems can form by assigning different roles to software developers or by choosing certain architectural patterns or development processes. Examples of such roles are software architects, system deployers, and component developers. Systems that not only consist of software introduce further groups of developers, e.g., electrical or mechanical engineers, that also produce artifacts that describe the system from physical viewpoints. To account for different perspectives and concerns, in short: for viewpoints, that different groups of developers have, the view-based paradigm has been established. Here, various views are defined that represent partial information of a system that is relevant from the viewpoint of the roles that different developer groups assume. With the usage of multiple models that constitute a system, these models often share information, which introduces redundancy, and the question of how to prevent that redundancy or keep redundant information consistent arises. Klare et al. [28] aim to answer that question by proposing the concept of *Virtual Single Underlying Model*s (V-SUMs) and presenting the Vitruvius approach. Aiming to combine "the advantages of synthetic and projective modeling" [28], the projective concept of a view as a dynamically generated model is combined with the synthetic concept of describing the system with multiple interrelated models instead of one. The concepts of projective and synthetic views are defined in the ISO 42010 standard [21]. With V-SUMs, developers only use views to modify the system. To be able to keep the system in a consistent state, *consistency preservation rules* (CPRs) are defined. They specify how consistency is preserved between models and are executed if changes are made to one model. Alongside view definition languages, languages that are specialized for consistency preservation play the key role in keeping consistency between the models that form the system description of a V-SUM.

While such languages exist and have been implemented for V-SUMs in the context of Vitruvius, there are also existing and mature [15, 13] general-purpose model-transformation languages that are able to express complex relations between models. One such concept is given by *Triple Graph Grammars* (TGGs) [32], which consist of context-sensitive graph production rule patterns that relate a pair of graphs by building a third graph representing the relation. These rules can be used for keeping two models consistent in an incremental manner via pattern matching.

TGGs allow for specifying descriptive rules that can concern any number of entities, and the graph approach allows for graphical visualization of these potentially complex relations. To use that advantage for consistency preservation in V-SUMs and examine the resulting approach, the following research questions are answered in this thesis:

1. How can TGG rules be applied in delta-based consistency preservation processes?
2. What kinds of consistency relations can be expressed with the developed concept?

To that end, the aim of this thesis is to investigate how TGGs can be used for consistency preservation in Vitruvius by researching how sequences of Vitruvius changes can be converted to sequences of TGG rule applications. This is done by elaborating a concept, and developing a prototype that implements that concept.

This concept, which is called *Backward Conversion Pattern Matching*, is explained in chapter 3. In addition to the process of converting TGG rules to something that is matchable to sequences of Vitruvius changes and the matching process, several additional process steps are necessary to make the concept realizable, such as handling deleting changes (section 3.6) or context matching (**??**). Details of the prototype implementation of the elaborated concept are described in extracts in chapter 4. The evaluation of the concept via the prototype is shown in chapter 5, as well as a discussion of the results. In chapter 6, a brief overview of work related to the thesis is given, mainly presenting other approaches to consistency preservation. Chapter 7 concludes by summarizing the work done in the thesis, the evaluation results and their discussion, and giving an outlook to future research related to this work.

2. Foundations

This chapter provides an overview of the theoretical and practical foundations of *Virtual Single Underlying Models* (V-SUMs), model-driven software engineering, model synchronization, and Triple Graph Grammars.

2.1. Model-Driven Software Engineering

Model-driven software Engineering (MDSE), or sometimes referred to as model-driven software development (MDSD), is an approach to software engineering that Kramer [29] describes as “a development paradigm in which models are used in an automated way for all development tasks”. Well-known approaches to this paradigm are the Object Management Group’s Model-Driven Architecture (MDA) approach [31] and the Unified Modeling Language [18], the latter being a language in which models can be described.

Definitions of what a model is are numerous. Stachowiak [33] provides three properties that define a model:

- mapping property: models always are models of *something*, they represent originals, which themselves can be models again.
- reduction property: models in general only comprise those attributes of the original that seemed relevant to model users or creators
- pragmatism property: models are not unambiguously assigned to their originals. They fulfill a purpose for certain subjects, within certain time intervals, and with limitations to certain conceptual or real operations.

Czarnecki and Helsen define the term model as “abstractions of a system or its environment, or both” [5]. Using this broad definition seems appropriate, since in the context of model-driven software engineering, program code is considered as models as well as formal descriptions of interaction like agent-based modeling or those of timing constraints (see [18]). A similarly broad definition is given by Caplat and Sourrouille. They define a model as “a representation of a system expressed in a given formalism or language” [3]. This formalism includes an abstract syntax, which is represented by one or multiple concrete syntaxes, and also semantics, which gives meaning to the abstract syntax and induces rules and conditions, sometimes called concrete semantics, which constrain what a well-formed model is and thus further restrict the syntax [16].

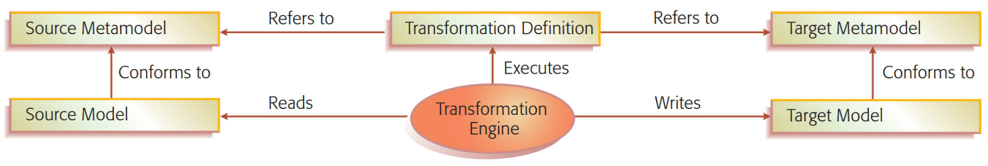

Figure 2.1.: Schematic example of a model transformation with one source and one target model.[5]

Eclipse Modeling Framework The *Eclipse Modeling Framework* [34] is a framework for modeling and code generation. It defines a metamodel called *Ecore*. In the implementation of Ecore, the concept of a *Resource* is used for persistence reasons. References between Resources are possible, and to realize that, *proxies* are introduced. An EObject can be a proxy or not. According to the EMF Javadoc [11], a proxy is "an object that is defined in a Resource that has not been loaded". Thus, the function of a proxy is to solve a chicken-and-egg problem by using placeholders while loading resources and resolving these later, when every required resource has been loaded.

2.1.1. Model Transformations

Model transformations play a key role in model-driven software engineering, as they enable specifying and automating relations between models. In general, model transformations map one or multiple source models to one or multiple target models. A transformation engine executes the transformation definition, which refers to the sources' and targets' metamodels, that define their abstract syntax. This process, as described by Czarnecki and Helsen [5], is schematically depicted in Figure 2.1 using the example of one source and one target model.

Semantic properties often cannot be distinctly related to a single model if a system is composed of multiple models that are developed together; however loosely coupled, a certain *semantic overlap* between these models occurs. Consistency preservation rules (CPRs) in the Vitruvius approach [28] realize some semantic considerations, in that case those concerning consistency, to relations between models, which are realized by model transformations.

2.1.2. Model Synchronization

In order to keep models that have a semantic overlap consistent, a change that occurs in one model has to be propagated to another model if the change concerns the semantic overlap. This synchronization process can be realized with model transformations that have certain features. In their model transformation classification approach, Czarnecki and Helsen identify features that can be used to characterize different approaches to consistency preservation. [5] The incrementality features are necessary properties for model synchronization, while directionality and tracing classify synchronization transformations.

Incrementality Incrementality comprises the features target incrementality, source incrementality, and preservation of user edits in the target. *Target incrementality* describes whether a transformation is able to propagate changes in the source model to an existing target model. Since consistency preservation requires propagating consistency-breaking changes in one model to another model with which it semantically overlaps, that is a necessary property of consistency preservation transformations. A transformation has the feature of *source incrementality* if it "minimizes the amount of source that needs to be reexamined by a transformation". This property is desirable from a performance point of view and with regard to potential information loss. *Preservation of user edits in the target* describes the ability of a transformation to apply source model changes to the target model while preserving modifications in the target model. A transformation that is target incremental but does not preserve user edits in the target is undesirable for consistency preservation purposes, since it may discard user edits and thus introduce information loss.

Directionality If a transformation can be executed in only one direction, it is called *unidirectional*, otherwise it is called *multidirectional*. A *bidirectional* transformation is a kind of multidirectional transformation that can be executed in two directions. In the context of model synchronization, where forward and backward transformations are required if both models are subject to user change, bidirectionality of a transformation reduces the definition effort of transformations and ensures that the forward transformation does the same as the backward transformation. If those are separate unidirectional transformations, ensuring that they apply the same definition of consistency has to be done elsewise, e.g., by the vigilance of the transformation developer or by deriving the unidirectional transformations from a shared consistency definition, like it is done in the Commonalities Language [27], which can thus be called bidirectional.

Tracing is described by Czarnecki and Helsen as a "runtime footprint of transformation execution". It is a relevant property for model synchronization transformations since it eases the identification of what model elements have to be changed in the target model.

2.2. Single Underlying Models

Single Underlying Models (SUMs) [1] are an approach to keeping a system consistent by using only one model to represent the system and defining views that are generated via model transformation and are used to access the model. This kind of approach can be called projective, since views are generated by "extraction from an underlying repository" [21]. This approach has the advantage, that no consistency-keeping has to be done between pairs of models because all information is contained in the SUM.

The benefit of total consistency comes with the drawback of a SUM being monolithic and thus, in larger projects, too complex to be handled by the methodologist responsible for creating the metamodel and view transformations. Wanting to keep the benefit of consistency while breaking complexity by using multiple metamodels internally, Klare et

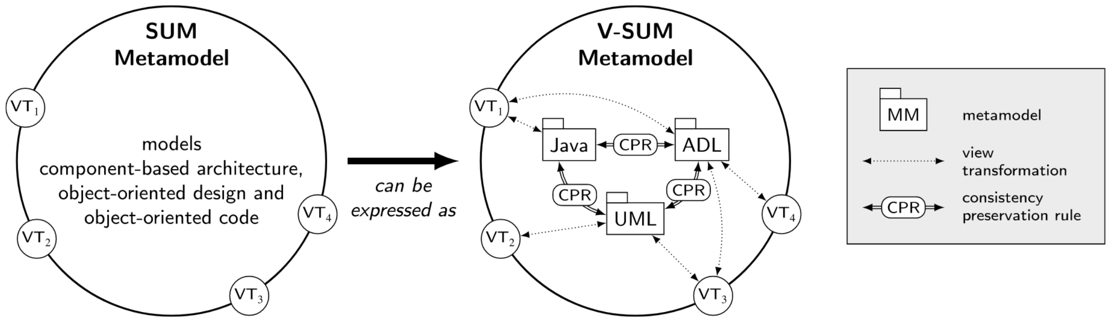

Figure 2.2.: A V-SUM compared to a SUM. From [28]

al. proposed the concept of a *Virtual Single Underlying Metamodel* V-SUMM, with V-SUMs describing the respective instances of the V-SUMM [28]. A V-SUM internally consists of multiple models but behaves like a SUM to the users who work with the views, since it is also only accessed via views, as shown in Figure 2.2. The model instances of these metamodels that share common information have to be kept consistent via model transformations.

2.3. The Vitruvius Approach

As an approach to realize a V-SUM, Klare et al. developed the Vitruvius approach [28]. What exactly is to be kept consistent is abstractly defined by the concept of *consistency preservation rules*, which is reified by the usage of languages like the Reactions Language and the Commonalities Language. The Vitruvius approach keeps track of consistency relations by letting CPRs use a correspondence model that helps identify what model elements are to be kept consistent.

2.3.1. Change Definition

Changes in Vitruvius are defined via a change metamodel, which itself is defined in Ecore, a meta-metamodel that closely resembles OMG's Essential Meta Object Facility (EMOF), which is supported by the Eclipse Modeling Framework (EMF) as an alternate serialization of Ecore [34, p. 39]. Ecore is supported for representing metamodels in Vitruvius [29].

This change metamodel consists of change meta-classes that concern atomic changes and compound changes that group atomic changes, marking that "these atomic changes occurred together" [29]. These compound changes are not explicitly modeled in Ecore, however, but are represented in the implementation.

The Vitruvius change metamodel incorporates a class hierarchy with multiple subconcepts of what a change can be. The root entity is an *EChange*. Multiple subclasses are additive or subtractive (by inheriting the respective abstract class), representing adding or deleting something from the model.

There are kinds of changes for

- EObjects: abstract classes for deletion, creation, and existence changing
- attribute and reference changes of EObjects: change classes that concern lists, single- or many-valued attributes and references.
- root EObjects that are not added by referencing another element

To map changes to model entities, the change classes are generically typed with Ecore classes, which represent the model elements to which changes are referring. As an example, a change to insert a new value into a many-valued attribute is modeled like in Figure 2.3.

```
InsertEAttributeValue<Element extends EJavaObject,
                      Value extends EJavaObject>
    -> InsertInListEChange<Element, EAttribute, Value>,
       AdditiveAttributeEChange<Element, Value>
```

Figure 2.3.: *InsertEAttributeValue* signature. InsertEAttributeValue inherits *InsertInListEchange* and *AdditiveAttributeEChange*

2.3.2. Correspondence Model

To trace which elements are to be kept consistent, Vitruvius uses a *correspondence model*. CPRs use that model to trace relations between elements that are to be kept consistent. This model basically consists of a simple abstract *Correspondence* class that has a tag for storing metadata to "distinguish different correspondences between the same element" [28] and two fields for left and right objects that are mapped to each other by being part of the same entity, as can be seen in Figure 2.4.

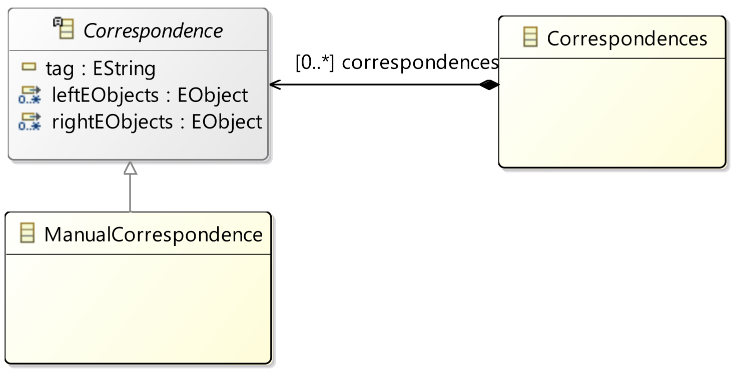

Figure 2.4.: The Vitruvius correspondence model. Generated from the current model definition [35]

2.4. Triple Graph Grammars

Schürr[32], who introduced Triple Graph Grammars, describes them as graph grammars or graph rewriting systems that rewrite three graphs in parallel while explicitly modeling inter-graph relationships in one of the graphs, the *correspondence graph*.

Graph Grammars *Graph Grammars* or "graph rewriting systems" [32] are sets of rules that are semantically similar to productions known from formal language grammars. These rules work in the following way: An object diagram, called *left-hand side*, represents a pattern that has to be found in a graph so that the rule can be applied. The *right-hand side* of the rule represents how that subgraph looks like after transforming the left-hand side via application of the rule. If all elements on the left-hand side can be identified with an element on the right-hand side, the rule is called a *non-deleting rule*. In such cases, each element on the left-hand side is identified with an element on the right-hand side; in Figure 2.5 this is depicted by arrows between the left- and right-hand sides. For deleting rules, only some elements on the left-hand side can be identified with elements on the right-hand side. To be able to maintain edges between the subgraph that is matched with the left-hand side and the rest of the graph, a *context graph* or *glue graph* is identified that contains the elements that are shared between the left-hand side and the right-hand side and remain unaffected by the transformation [6].

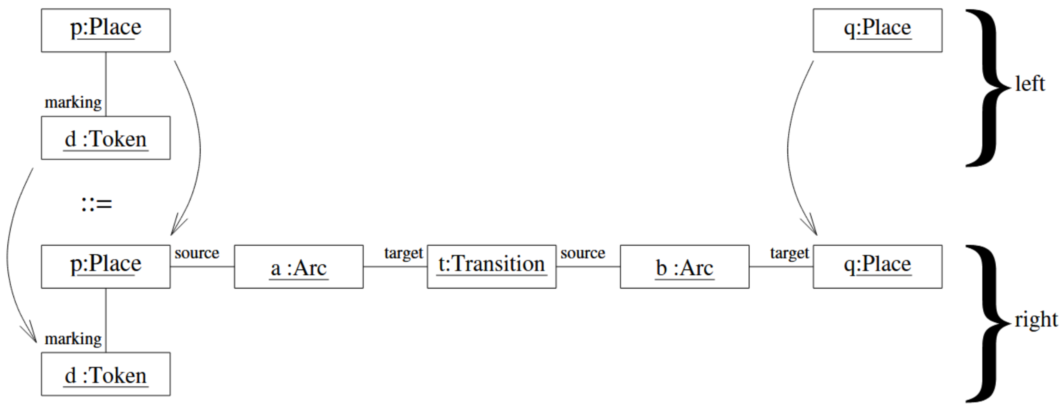

Figure 2.5.: A non-deleting graph grammar rule. [26]

While Graph Grammars are described as "well-suited for the description of complex transformation or inference processes on complex data structures" [32], they are limited to specifying in-place modifications within only one graph.

Triple Graph Grammars *Triple Graph Grammars* (TGGs) were introduced by Schurr [32] to extend this capability to model relationships between two (or more) graphs, thus allowing for simultaneous evolution of models while accounting for their relationship. This is achieved by an explicitly modeled correspondence graph and a rule set that maps one model triplet, consisting of source, target, and correspondence graph, to another triplet in a similar way, as described for graph grammars in general. The first triplet (left-hand side) again represents the pattern to be found in the graph, which constitutes of the source, target and correspondence graphs. The second triplet (right-hand side) represents that subgraph after transformation.

Model Synchronization with TGGs This way, TGG rules also can be used to describe what has to be changed in a target graph when a change in the source graph occurs. If such a change is matchable to the left-hand side and right-hand side of a TGG *source rule*, meaning the part of the TGG rule that concerns the source graph, the respective target rule can be applied to the target graph. The correspondence graph directly references the subgraph in the target graph to which the target rule has to be applied. That process, from change occurring in the source to applying the matching change in a target graph, represents the application of the TGG rule on the triple graph. Since the labeling of the source and target graphs as “source” and “target” is arbitrary, TGG rules can be described as bidirectional synchronization transformations.

Representing the deletion and modification of elements with TGGs can be done by inversely applying TGG rules, by rolling back rule applications, removing the rule application that created the element one wants to delete, and reapplying rule applications. Both of these approaches can introduce information loss in the target model if a modification is represented. In the inversion case, a modification would be represented as an inverse application of a TGG rule, followed by a re-application of the same rule, but not inversed and with other parameters or another rule. Since the inverse application deletes model elements in the target model that might be enriched with other model elements that are not covered by the TGG rules, information loss can occur. The same problem occurs with the rollback approach. In certain situations, only inverting a rule application is not sufficient to represent a deleting change, e.g., when two model elements were generated together, in the sense that both creations are represented by one rule application, and later, only one of these is to be deleted. Then, the one that is not to be deleted, hinders the mere application of the inverse rule. In that case, it either would have to be re-created after inversion, or the rollback approach would need to be done. So both approaches to handling deleting and modifying changes may introduce information loss in the target model. Having reviewed different approaches for handling these situations, Fritsche et al. come to the conclusion that an approach that “avoids unnecessary information loss, is proven to be correct, and is efficiently implemented”, is still missing [14]. They present *short-cut rules* [15, 14], which are an approach to cope with the problem of information loss by combining the inverse rule and the other rule. Nodes that would be deleted in the first and re-created in the second rule application are found out and retained in the left-hand sides and right-hand sides of the resulting short-cut rule. While the original short-cut rules were created at compile time, Fritsche et al. recently presented higher-order short-cut rules [13] which are generated at runtime to cover more rule combinations and thus further decrease information loss in the target model. Short-cut rules can be viewed as an effort to improve incrementality features of TGGs for model synchronization, as described in subsection 2.1.2.

2.5. eMoflon::IBeX

eMoflon::IBeX [38] is a tool suite for TGGs which implements model generation, transformation, and synchronization, as well as consistency checking. It encompasses a strategy to allow for information-preserving model changes in many cases, which is non-trivial and reduces rule-writing overhead. This is done by synthesizing *short-cut rules* [14, 13] that combine one rule that is to be revoked with another rule that replaces it. The resulting rule preserves the entities that would otherwise have been deleted and re-created. Especially if the rule invocation has been done relatively early, this becomes crucial if one does not want to lose information in the deleted-and-recreated entities that is not part of the TGG modeling and can thus not be recreated together with the entities.

Correspondences in eMoflon::IBeX are modeled similarly to, but differently from how Vitruvius [28] models them (see subsection 2.3.2). In eMoflon::IBeX, the *schema* defines a correspondence metamodel between two metamodels, whose instances are to be kept consistent. See Figure 2.6 for an example schema definition and Listing 2.1 for an example usage of this schema in a rule definition. This correspondence metamodel is then used in rule definitions to model the actual correspondences between entities of source and target. The correspondences are modeled similarly to how Vitruvius does it, correspondence types are given a name and source and target EClasses. One noteworthy difference is that in Vitruvius, the user does not pre-define an explicit correspondence metamodel, which is used to manage correspondences in the CPRs, but instantiates the existing correspondence metamodel, which is shown in Figure 2.4. In the *Reactions* language, described in chapter 6, this metamodel is part of the language.

```
#using Java2Uml.*
#using AttrCondDefLibrary.*

#abstract #rule AttributeToProperty #with Java2Uml
  #source {
    classifier:java.classifiers.Classifier
    ++field:java.members.Field
  }
  #target {
    umlClassifier:uml.Classifier
    ++property:uml.Property

  }
  #correspondence {
    classToUmlClass:JavaClassifierToUmlClassifier {
      #src->classifier
      #trg->umlClassifier
    }
    ++attributeToProperty:JavaAttributeToUmlProperty{
      #src->field
      #trg->property
    }
  }
  #attributeConditions { eq_string(field.name, property.name) }

```

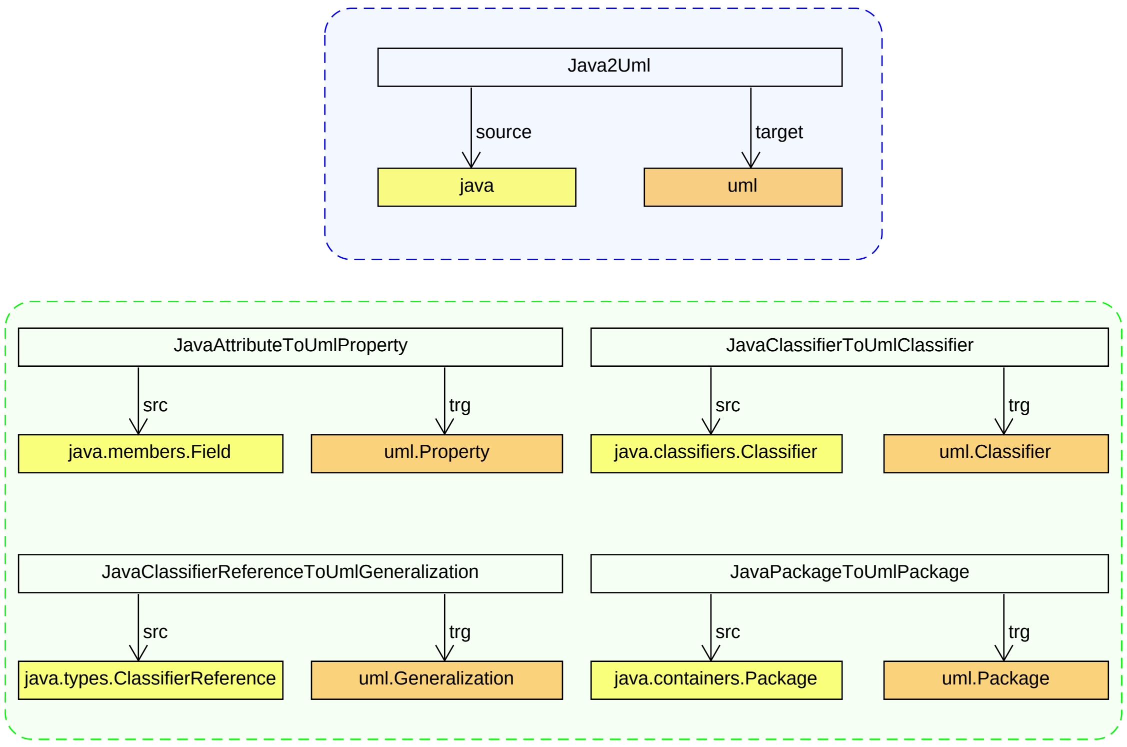

Figure 2.6.: Schema definition in eMoflon::IBeX: The schema and its source and target metamodel are specified (top), as well as correspondence types (bottom) that refer to EClasses of the respective source and target metamodels.

```
#rule ClassAttributeToProperty #extends AttributeToProperty #with Java2Uml
  #source {
    classifier:java.classifiers.Class {
      ++ -members->field
    }
  }
  #target {
    umlClassifier:uml.Class {
      ++-ownedAttribute->property
    }
  }
...
```

Listing 2.1: Rule defition in eMoflon::IBeX: correspondence relationships can only refer to correspondence classes defined in the *schema* (see Figure 2.6). Rules can be abstract and concrete. Inheritance is possible.

2.5.1. Attribute Conditions

Ecore differentiates between *EObjects* and *EDataTypes*. Since EDataTypes are not part of the model graph, but fields of EObjects, they cannot be represented in synchronization by plain TGG rules. To be able to synchronize attributes, *attribute conditions* are used [38, 8]. Attribute conditions define constraints between attributes of the source and target model that must be fulfilled for a TGG rule match to be valid. In eMoflon::IBeX, the user can define for which attribute states condition checking or enforcement should be applied, and the condition checking/enforcement procedure can depend on that state combination. An attribute can be in *free* or *bonded* state, depending on whether a value has been assigned to it. That way, attribute conditions can be used to set free attributes (e.g. in the target model) based on other, bonded attributes (e.g. in the source model), but also to assert a consistency relationship between bonded attributes. There is a predefined library for common attribute conditions, such as string equality. Additionally, custom attribute conditions can be defined by the user.

2.5.2. Synchronization Process

eMoflon::IBeX implements the synchronization process described by Fritsche et al. [14], which is based on a synchronization algorithm that uses an *incremental pattern matcher* that provides currently applicable and broken TGG rule matches, from which the synchronization chooses what to apply. In Figure 2.7, the synchronization process is shown in a simplified version. By calling the pattern matcher on a possibly unsynchronized triple graph, new TGG rule matches, called “forward matches”, are calculated, as well as matches that have existed before calling the pattern matcher and have become invalid (“broken”). If no forward or broken matches were calculated, the algorithm terminates. Else, if forward matches are present, one is chosen and applied, and the process is started anew. If no forward matches are present, but broken matches are, they are attempted to be repaired via short-cut rules (see section 2.4, [15, 14, 13]). Remaining broken matches that could not be repaired, are revoked, freeing nodes of the triple graph and possibly enabling new forward matches. Thus, again, the process is started anew, after the revoking step.

2.6. Detection of Complex Changes

Identifying and choosing complex change patterns in sequences of atomic changes has been researched by Khelladi et al. [25] in the context of metamodel evolution, aiming to better and more coarsely grained represent the intention of users than with the atomic changes “add, delete, and update elements”. To deal with the challenge of complex change patterns overlapping after identifying these patterns in a sequence, they introduce three heuristics, which are briefly explained in the following:

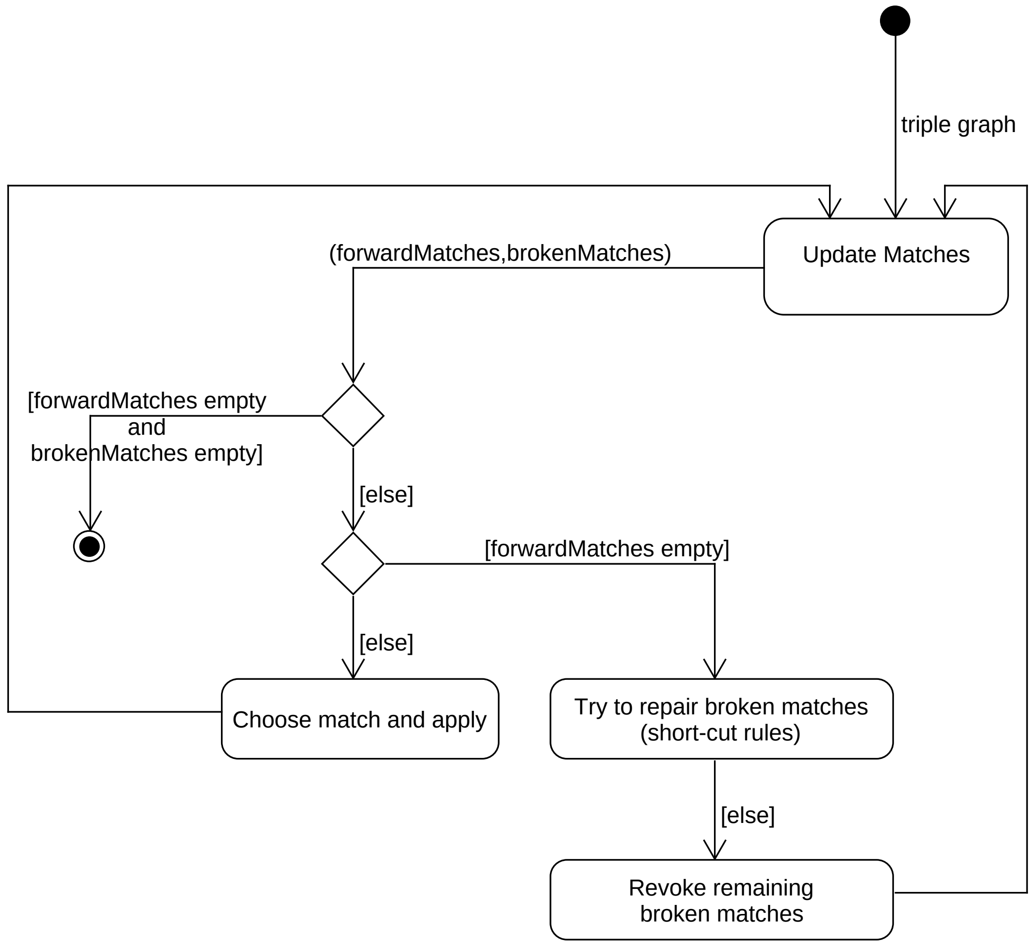
triple graph
Update Matches
(forwardMatches,brokenMatches)
[forwardMatches empty and brokenMatches empty]
[else]
[forwardMatches empty]
[else]
Choose match and apply
Try to repair broken matches (short-cut rules)
[else]
Revoke remaining broken matches

Figure 2.7.: Simplified activity diagram based on the synchronization process in *eMoflon::IBeX* based on the synchronization algorithm presented in [14]. Arrow labels in `[brackets]` indicate control flow, other labels indicate data flow.

Containment heuristic For applying the *containment heuristic*, each complex change pattern is assigned a *priority* based on the length of the longest containment path from the pattern under consideration to another pattern it contains. As an example, let P_1, P_2, P_3 be three complex change patterns with the containment relation being $P_3 \subset P_2 \subset P_1$. Then, the priorities are as follows: $priority(P_1) = 1, priority(P_2) = 2, priority(P_3) = 3$. In applying the containment heuristic, complex change patterns with higher priority are preferred over patterns with lower priority.

Distance of a complex change The distance heuristic is defined as follows: $Distance = \frac{S_{CC}-1}{EP-SP}$. S_{CC} is defined as the size of the complex change, and EP and SP refer to the end position and the start position of the detected complex change in the change sequence. This heuristic is between 0 and 1 and measures how widespread a pattern is relative to its size. A detected complex change that occurs in the change sequence “without interruption” by other changes has a distance of 1. Higher distances are preferred to lower distances when choosing between overlapping patterns.

Solving overlapping rate The *solving overlapping rate* heuristic aims to "minimize the number of overlapping changes" by removing candidates. Thus, the higher the reduction of remaining overlapping changes that is caused by the removal of a complex change from the candidate list, the higher the solving overlapping rate. Formally, it is defined as $SolvingOverlappingRate = 1 - \frac{N_{LOCC}}{N_{OCC}}$, N_{LOCC} being the number of overlapping changes that would remain if the currently considered complex change were disregarded, and N_{OCC} being the total number of overlapping complex changes.

3. Concept

To be able to utilize TGGs for defining consistency preservation rules in Vitruvius, a transformation process from sequences of atomic changes from the Vitruvius change metamodel (see section 2.3) to applications of TGG rules has been developed, and a prototype has been implemented.

In Vitruvius, all changes are atomic or composite, with composite changes being a sequence of atomic changes. "atomic" is defined by Klare et al. [28] as affecting "only one element value". TGG rule applications differ from Vitruvius changes insofar as pattern applications can affect more than one model element and are intended to be able to express complex relationships in consistency definitions in the change model.

In the context of using that ability of TGGs for Vitruvius, this gives rise to the question of how those atomic Vitruvius changes can be accurately mapped to complex TGG rule applications and the question of how TGG rules can be represented to enable this mapping. In this thesis, these questions were investigated by developing and researching existing strategies for correctly detecting complex patterns in atomic change sequences and applying them to the given use case and/or developing new strategies.

Figure 3.1 gives an overview of the transformation process concept. A sequence of atomic Vitruvius changes to a source model, recorded by a change monitor or derived from state differences (both being existing Vitruvius functionality) is given, as well as a source and a target model. The process transforms the given sequence to a sequence of TGG source rule applications. The source rule definitions are derived from TGG rule definitions. The source rule application sequence is used to synchronize the target model. Since TGG source rules map to target rules because both are derived from a single TGG rule, the source rule application sequence can be mapped to a target rule application sequence, which is then used by the TGG engine to synchronize the target model. For full Vitruvius integration, the target rule application sequence can be transformed to a sequence of atomic Vitruvius changes to the target model, which is not done by the TGG engine but by the conversion process. However, that step is not covered by the implemented prototype.

In the remainder of this chapter, the developed concepts and algorithms to realize the process of converting Vitruvius change sequences to TGG rule applications are described.

The foundational concept is called *Backward-Conversion Pattern Matching*. The first section (section 3.1) describes the core of this concept, and the sections section 3.3 and section 3.4 its realization in detail. Further sections consider concepts and algorithms that solve necessities that Backward-Conversion Pattern Matching introduces if one does not want to fall back to using a solely graph-based pattern matcher.

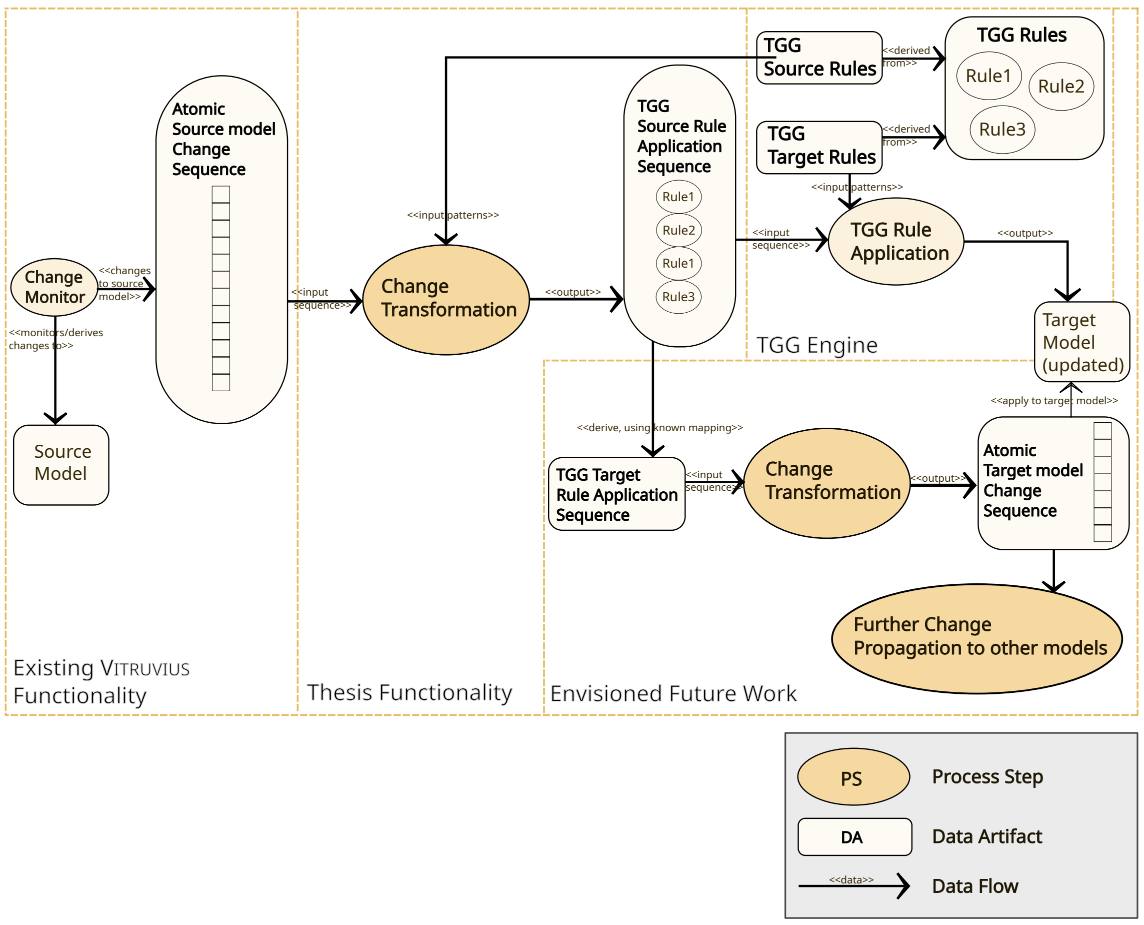

Figure 3.1.: An Overview of the transformation process concept.

3.1. Backward-Conversion Pattern Matching: Overview

In this thesis, the foundational concept for solving the problem of generating TGG rule application sequences out of VITRUVIUS change sequences is to transform the problem to the Vitruvius change space and solve it there. It involves converting TGG patterns to template sequences of atomic VITRUVIUS changes and matching those template sequences in a change sequence given by VITRUVIUS. In the following, this concept is briefly outlined.

Imagining the whole process as a pipeline, depicted in Figure 3.3, the conversion of TGG rules to templates that are mappable to change sequences, this step can be seen as a step backwards, thus it is called *backward conversion*. Backward conversion identifies the *green* nodes and edges (see Figure 3.2) of the source or the target side of a TGG rule, maps them to VITRUVIUS *EChange* classes, and builds a data structure that preserves the graph structure implicitly and is mappable to changes from a change sequence. Section 3.2 describes the algorithm and the data structure in more detail.

Upon having converted TGG rules to change sequence templates, *green matching* can be performed. This is called green matching because only the *create* nodes and edges, also

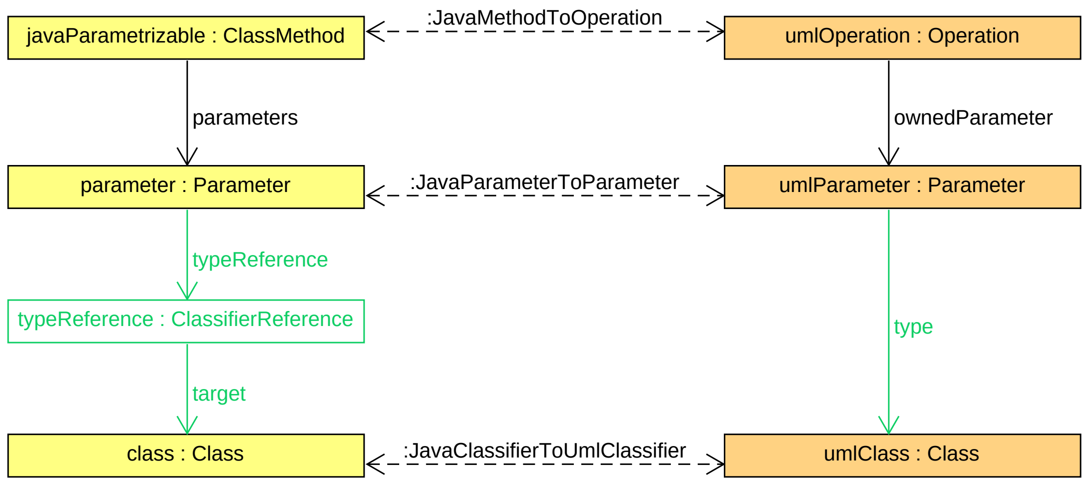

Figure 3.2.: Running Example *MethodClassParamTypeToParamType*: A triple graph grammar rule describing a consistency relationship between typing of method parameters in Java and UML. Green nodes and edges indicate the right-hand side of the rule, i.e. what is added to both models.

called green nodes and edges, that a TGG rule contains are considered here. The reason for not also considering the (black) context nodes at this point is that context is partly existing prior to the pattern matching process, but also partly arising in the process. So, matching context prematurely would prevent finding matches whose context is created at the time of pattern matching. In section 3.3, the process of invoking change sequence templates upon change sequences and covering the sequences with such invocations is described.

After receiving change sequence template applications from green pattern matching, each of these is tried to be matched fully, meaning that, in addition to the green nodes already matched, black nodes are matched against model elements, too. Because applying a black-matched TGG rule application to the target model can enable other change sequence template applications to be matchable, this process is done repeatedly until there are no more new TGG rule applications. Section 3.4 describes the algorithm of completing a green match by trying to map the context nodes that remain unmatched after green matching. This requires a graph-based approach that traverses both source model and TGG rule alongside, recursively extending the mapping of TGG rule node to source model node that was created by green matching.

Between black pattern matching and applying the fully matched TGG rule application to the target model, pattern selection is performed, which is described in 3.5. This is necessary because there might be multiple rule applications covering one or more EChanges, and only one of those can be applied. This choice has to be made after black pattern matching, and thus, possibly multiple times, to maximize coverage and not select-out pattern applications that fail black pattern matching.

To deal with subtractive changes, red pattern matching, described in section 3.6, is performed. Subtractive changes invalidate previously applied TGG rule applications, but this might

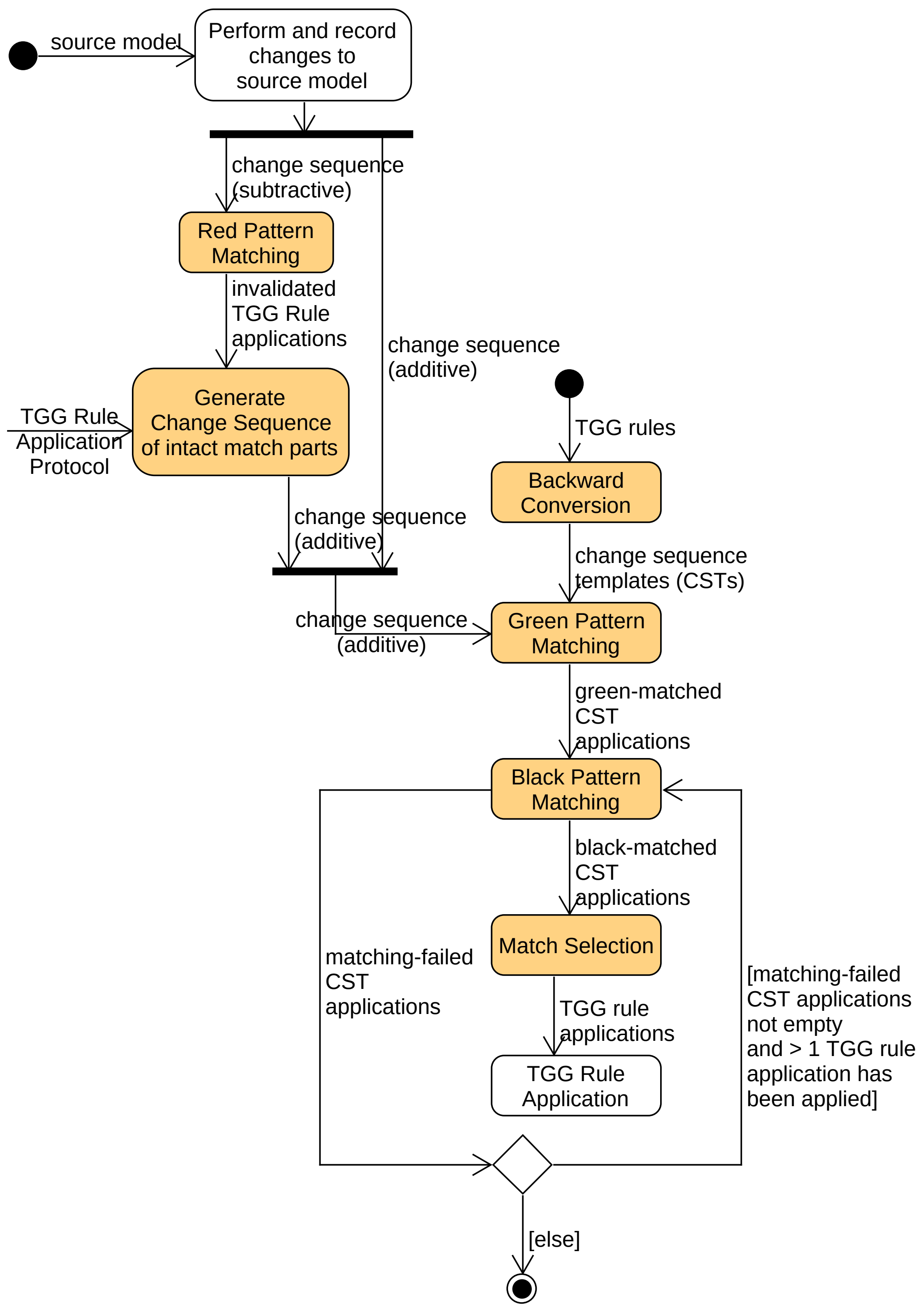
source model
Perform and record changes to source model
change sequence (subtractive)
Red Pattern Matching
invalidated TGG Rule applications
TGG Rule Application Protocol
Generate Change Sequence of intact match parts
change sequence (additive)
change sequence (additive)
TGG rules
Backward Conversion
change sequence templates (CSTs)
change sequence (additive)
Green Pattern Matching
green-matched CST applications
Black Pattern Matching
black-matched CST applications
Match Selection
matching-failed CST applications
TGG rule applications
TGG Rule Application
[matching-failed CST applications not empty and > 1 TGG rule application has been applied]
[else]

Figure 3.3.: Activity Diagram, giving an Overview of data flow in the Backward Conversion Pattern Matching concept, from change sequence retrieval to applying TGG rules in the target model. The orange nodes mark the process steps of the concept. Arrow labels in `[brackets]` indicate control flow, other labels indicate data flow.

also result in other EChanges being covered by a rule application no more. Thus, first detecting which TGG rule applications are now invalid, these "freed" EChanges are detected, re-created, and added to the other additive changes given by the currently recorded changes to the source model.

3.2. Backward Conversion: Making TGG patterns matchable to change sequences

To be able to base the pattern matching of TGG rules to model changes on Vitruvius change sequences, in this section, the idea and conceptual realization of *backward conversion* of TGG rules is presented. The key idea is to represent the structure of a TGG rule in the form of a template that is more easily matchable to a sequence of atomic changes than trying the same with a graphically represented TGG rule would be. To that end, the idea of extracting the change information of green nodes and edges and building a data structure that represents these changes and their interrelation in a form that resembles a collection of Vitruvius *EChanges*, that are not initialized with actual elements but with placeholders that bind these changes together was created and is elaborated in the following.

For matching TGG rules to sequences of changes that have occurred in a model, only that side of the TGG rules that has the same metamodel as the model in which the changes have taken place is needed for backward conversion and green matching. To be able to match TGG rules to sequences of changes of the Vitruvius change metamodel(*EChanges*), a custom data structure is defined that serves as the output of the *Backward Conversion* process. In the following, this data structure is described. After that, the algorithm that parses a TGG rule into this data structure is presented. Figure 3.4 shows how the *Change Sequence Template* data structure is constructed, and in Figure 3.5, an instance example can be seen.

Change Sequence Templates *Change Sequence Templates* represent one converted rule. They consist of EChange wrappers, that hold the type of EChange they represent plus one or multiple placeholders for actual EChanges. *EChange Wrappers* each represent one atomic EChange class of the Vitruvius change metamodel that might occur in a change sequence. They also represent one green node or edge in the rule pattern of the change sequence template they belong to. Depending on the type of EChange that is represented, an EChange wrapper holds different numbers of *Placeholders*. This is exemplarily shown in Figure 3.5. Each placeholder stands for an actual object that is concerned by possibly more than one EChange. If more than one EChange concerns the same EObject, that EObject will still only be represented by one placeholder, which then occurs in multiple EChange wrappers. In that way, placeholders ensure that the graph structure of the TGG rule is retained throughout the conversion process.

The process of converting a TGG rule to generate a change sequence template is realized via a depth-first search through the TGG rule graph.

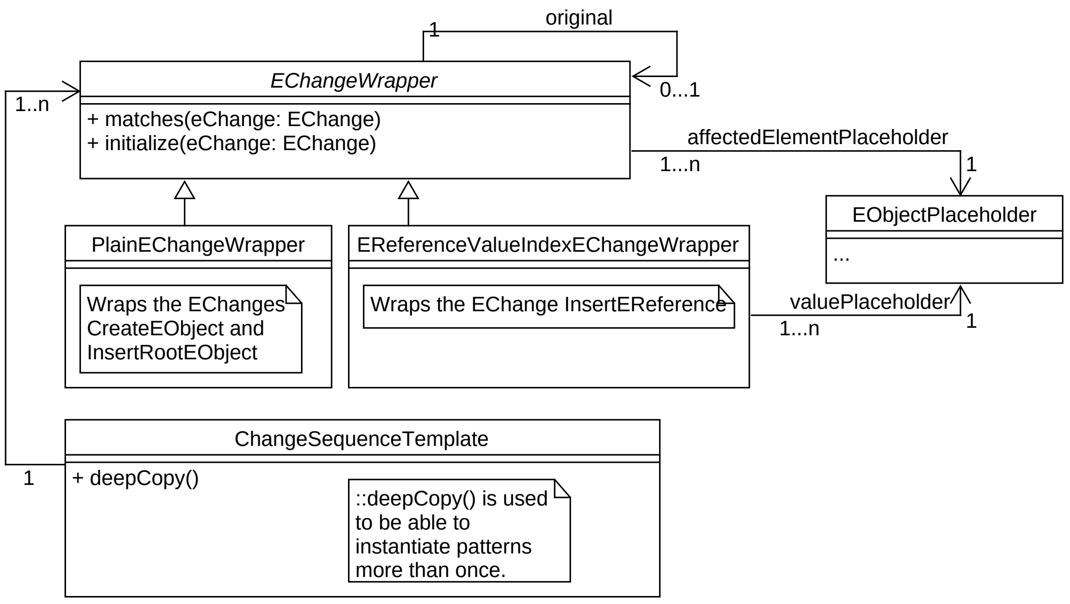
original
1
0...1
EChangeWrapper
+ matches(eChange: EChange)
+ initialize(eChange: EChange)
1..n
affectedElementPlaceholder
1...n
1
EObjectPlaceholder
...
PlainEChangeWrapper
Wraps the EChanges CreateEObject and InsertRootEObject
EReferenceValueIndexEChangeWrapper
Wraps the EChange InsertEReference
valuePlaceholder
1...n
1
ChangeSequenceTemplate
+ deepCopy()
1
::deepCopy() is used to be able to instantiate patterns more than once.

Figure 3.4.: Change Sequence Templates, EChangeWrappers and EObjectPlaceholders are used to represent a TGG rule in a way that makes it more conveniently matchable to a sequence of atomic VITRUVIUS changes. EObjectPlaceholders are contained in possibly more than one EChangeWrapper, sometimes as *affectedElement*, sometimes as *value*. To be able to apply Change Sequence Templates more than once, they are copyable.

The algorithm visits the green nodes and edges of the rule, and parts of its direct neighborhood consisting of context nodes that are connected to green nodes via green edges. In that case, the context node is modified, so it must be taken into consideration.

For green nodes, one $PlainEChangeWrapper$ is created based on the $EClass$ that the green node represents and typed as a $CreateEObject$. Further, the wrapper is given a placeholder, that is created if there is none already existing for the node. If a green node does not have any incoming context or create edges in the same domain (source or target, depending on which is currently looked upon), that is interpreted as the creation of a root node in the model, so a second $PlainEChangeWrapper$ is created and typed as an $InsertRootEObject$. Placeholder and $EClass$ of the affected EObject stay the same as with the other wrapper.

In the case of a green edge, an $EReferenceValueIndexEChangeWrapper$ is created, typed with $InsertEReference$. It is given the EClass that the source node of the edge represents, the placeholder of the source node, the EReference that the edge represents, and a placeholder for the value that is inserted into the reference.

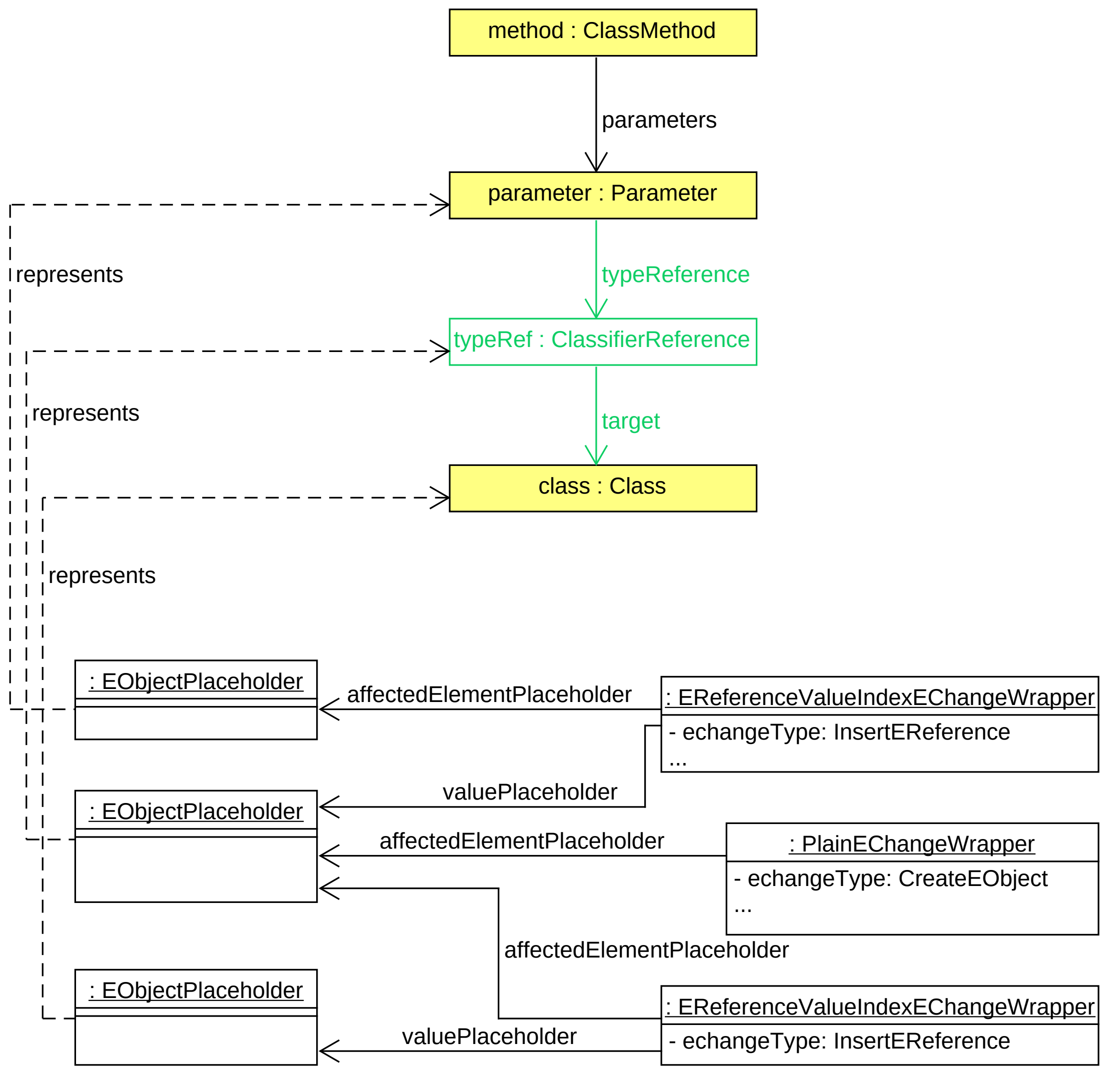
method : ClassMethod
parameters
parameter : Parameter
typeReference
typeRef : ClassifierReference
target
class : Class
represents
represents
represents
: EObjectPlaceholder
: EObjectPlaceholder
: EObjectPlaceholder
affectedElementPlaceholder
valuePlaceholder
affectedElementPlaceholder
affectedElementPlaceholder
valuePlaceholder
: EReferenceValueIndexEChangeWrapper
- echangeType: InsertEReference
...
: PlainEChangeWrapper
- echangeType: CreateEObject
...
: EReferenceValueIndexEChangeWrapper
- echangeType: InsertEReference

Figure 3.5.: Running Example *MethodClassParamTypeToParamType*. The colored graph at the top shows the source graph of a triple graph grammar rule. Bottom graph: Instance diagram of the converted source graph in the Change Sequence Template data structure.

3.3. Green Pattern Matching: Additive Changes

After Change Sequence Templates are created, a change sequence can be matched against those templates to detect the complex patterns that form the precondition for applying the TGG rule, i.e. consistency preservation rule. In this section, the algorithm to partly and fully invoke Change Sequence Templates by traversing the change sequence is given and described. A match found by this algorithm does not automatically mean that it is really applied, since context matching (see section 3.4) and pattern selection (see section 3.5) sort out matches.

The algorithm can be seen in 3.1 and works in the following way: For each change of the given change sequence, it generates all possible partly initialized invocations of change sequence templates for that EChange. If some are duplicates of template invocations generated in previous loop iterations, these are discarded. The remaining invocations are globally stored to keep the result and for duplicate detection. Then, each of these template invocations is tried to be fully matched against the other EChanges of the change sequence. To that end, each EChangeWrapper contained in the currently treated template invocation is tried to be matched with each EChange that comes into question until a match is found. These EChange candidates are determined by whether their EChange type matches the EChangeWrapper's type. Matching is performed by checking whether the types of the EObjects and, e.g., the EReference contained in the EChange match the types expected in the EChangeWrappers.

In the following, helper functions that are not written in pseudocode are described:

REMEMBERWRAPPERSINVOKEDWITHECHANGE takes an EChange and a set of templates partly initialized with the given EChange. For each partly initialized template invocation, the function finds the EChangeWrapper that has been initialized with the given EChange and remembers, globally, that the original (see Figure 3.4) of that EChangeWrapper has been invoked with the given EChange. This allows for removing duplicates with REMOVEDUPLICATETEMPLATEINVOCATIONS.

REMEMBERWRAPPERINVOKEDWITHECHANGE takes an EChange and an EChangeWrapper that has been initialized with the given EChange. It remembers, globally, that the original (see Figure 3.4) of the given EChangeWrapper has been invoked with the given EChange. This allows for removing duplicates with REMOVEDUPLICATETEMPLATEINVOCATIONS.

REMOVEDUPLICATETEMPLATEINVOCATIONS takes an EChange and a set of templates partly initialized with the given EChange. For each partly initialized template, the function checks whether the *original* (see Figure 3.4) of the EChangeWrapper in the template that contains the given EChange has already been invoked with the given EChange and removes the thus detected duplicates from the set.

ChangeSequenceTemplateSet::INVOKETEMPLATESBYECHANGE takes an eChange and returns all possible invocations of change sequence templates contained in that set. This means each template that contains one or more EChangeWrappers that match the eChange is invoked. Invoked means, the template is *deep copied*, which means that all EChangeWrappers and placeholders are copied, too. In the EChangeWrappers' case, the *original* reference is set to the original EChangeWrapper. If a template can be invoked more than once because multiple contained EChangeWrappers match the given eChange, multiple invocations of the same pattern, but each with a different EChangeWrappers initialized, are returned.

Algorithm 3.1 Pattern Matching with Change Sequence Templates

```
$changeSequenceTemplateSet \leftarrow$ CONVERTTGGRULES(...)
$allInvokedTemplates \leftarrow$ HASHSET
function GETGREENMATCHES($changeSequence : Sequence < EChange >$)
    for all $eChange : changeSequence$ do
        $partlyInitializedTemplates \leftarrow changeSequenceTemplateSet$
                .INVOKETEMPLATESBYECHANGE($eChange$)
        REMOVEDUPLICATETEMPLATEINVOCATIONS($eChange$,
                                        $partlyInitializedTemplates$)
        REMEMBERWRAPPERSINVOKEDWITHECHANGE($eChange$,
                                        $partlyInitializedTemplates$)
        $allInvokedTemplates.addAll(partlyInitializedTemplates)$
        for all $partlyInitializedTemplate : allInvokedTemplates$ do
            if $\neg$TRYTOFULLYMATCH($partlyInitializedTemplate$) then
                $allInvokedTemplates.remove(partlyInitializedTemplate)$
            end if
        end for
    end for
end function

function TRYTOFULLYMATCH($partlyInitializedTemplate : ChangeSequenceTemplate$)
    $uninitializedWrappers \leftarrow partlyInitializedTemplate$
                                    $.uninitializedEChangeWrappers$
    for all $eChangeWrapper : uninitializedWrappers$ do
        $wrapperIsInitialized \leftarrow false$
        $eChangeCandidates \leftarrow eChangesByEChangeType$
                                    $.get(eChangeWrapper.eChangeType)$
        for all $eChangeCandidate : eChangeCandidates$ do
            if $eChangeWrapper.matches(eChangeCandidate)$ then
                $eChangeWrapper.initialize(eChangeCandidate)$
                $wrapperIsInitialized \leftarrow true$
                REMEMBERWRAPPERINVOKEDWITHECHANGE($eChange$,
                                            $eChangeWrapper$)
            end if
        end for
        if $\neg wrapperIsInitialized$ then
            return false
        end if
    end for
    return true
end function
```

3.4. Black Pattern Matching: Context

As can be seen in Figure 3.2, a TGG rule does also consist of context nodes, also called black nodes. Since it is necessary to match one whole side of the pattern to be able to apply the green changes on the other side, the pattern-matching algorithm described in section 3.3 is not sufficient. Thus, in this section, a context-matching algorithm is described that starts from a successfully matched *change sequence template*. In particular, this means that this change sequence template provides a bijective mapping from green *TGGRuleNode*s to *EObjects*, if only that one match is looked upon. The goal of black pattern matching is to expand this mapping so that it also contains all black context nodes.

To reduce complexity in the algorithm design, a data structure called *MapStack* is defined. This data structure holds the aforementioned mapping from *TGGRuleNode*s to *EObjects*, but also provides stack functionality to enable the matching algorithm to branch and try different mappings. If the algorithm comes to a branching situation, where multiple EObject candidates are possible, one after another is tried, and, in case of failure, the MapStack is set back to the state before the branching, so the next branch can be tried. The following methods of MapStack are of interest:

MapStack::putPush takes a *key* and a *value* and stores their mapping. It also remembers the key in its internal stack.

MapStack::get takes a *key* and returns the *value* to which the key is mapped.

MapStack::removePopUntil takes a *key* and pops all keys until and including the given key from the stack. It removes them from the internal key-value mapping and returns all those keys.

The basic idea of the context matching algorithm is to start a depth-first search at each green node, in both incoming and outgoing directions, and from there, move simultaneously in both the TGGRule graph and the graph given by the model. Both are "sewn together" where the green nodes are mapped to EObjects, so that is an ideal starting point. Talking from a TGG perspective, the algorithm does not stay in the source or target graph but behaves differently in the source and the target. Without loss of generality, let *source* be the TGG graph domain where the green pattern matching has taken place for this section. In the target graph, the algorithm does ignore green nodes or edges, because those are precisely the ones that are to be created in the consistency-keeping process. Nonetheless, context nodes have to be matched in the target graph. To further reduce complexity, the algorithm never traverses green edges; it simply starts at each green node and only searches the black nodes and edges reachable from each of the green source nodes. visitNode is called on each green node from the rule, which visits each incoming and outgoing edge, failing early if matching failed on one edge.

visitOutgoingEdge handles the target node of a rule edge and tries to match it to an EObject from the model. To that end, each EObject referenced by the EObject mapped to the source node and typed according to the EReference provided by the given *outgoingEdge* is tried via tryAllMatchingCandidatesFor (see 3.3). Since outgoing edges always stay in the domain, the target rule nodes can either be context or create nodes. Create nodes are ignored, since they are handled by contextMatches, the starting point of the black matching algorithm.

tryAllMatchingCandidatesFor (see Algorithm 3.3) takes a rule node and a set of EObject candidates that are to be tried out to be mapped to the given rule node. The caller ensures that all EObject candidates are mappable to the rule node, so this function's only task is to try out the candidates by setting them as the rule node's mapping and recursing. After each failed recursion, the state of the mapping and the marking of visited nodes is reset to the state before the recursion.

visitIncomingEdge (see Algorithm 3.2) handles the source node of a rule edge and tries to match it to an EObject from the model. This is more complicated than visiting outgoing edges, because incoming edges can either be from the same domain or from the correspondence graph. If the incoming edge is of the same domain as its target node, the procedure is similar to visitOutgoingEdge, but not quite the same. Incoming edges in the Ecore model can either be *containments* or *cross-references*. Cross-references have to be found first, which is what findAllEObjectsReferencing does. In that case, again, there can be multiple candidates, and thus tryAllMatchingCandidatesFor (Algorithm 3.3) is used to try them out. If the incoming edge is not of the same domain as its target node, its source node must be of the TGGs correspondence graph. In that case, the correspondence node instance is tried to be found in the TGG *protocol* and mapped to the source node (correspondence node). Also, via the correspondence node instance, the related EObject in the other domain is found and mapped to its respective rule node. That can only be a context node, since for a correspondence node instance to exist, its source and target node instances must also exist. From that corresponding rule node, that is from the other domain, the recursion is continued.

3.5. Match Selection

As there might be rule matches that overlap in what they cover from the change sequence, a selection of matches becomes necessary to choose which rule match should be applied. As described in section 2.6, Khelladi et al. [25] introduced heuristics for that which are modified and extended in this thesis. The containment level heuristic is modified to be usable with a dynamic ruleset, the *distance heuristic* is used as-is, and a third heuristic called *max coverage* is added to account for one of this work's goals, which is preserving consistency . The combination of these heuristics, the modification of the containment level heuristic, and the idea of the max coverage heuristic is described in the following.

As envisaged by Khelladi et al. [25], this concept uses more than one heuristic at a time: The *containment level heuristic* is used first. It is modified to detect the containment level dynamically, not by pre-calculated levels, since that is not possible with a dynamic number and structure of TGG rules. In Figure 3.6, an application example for the containment level heuristic is shown.

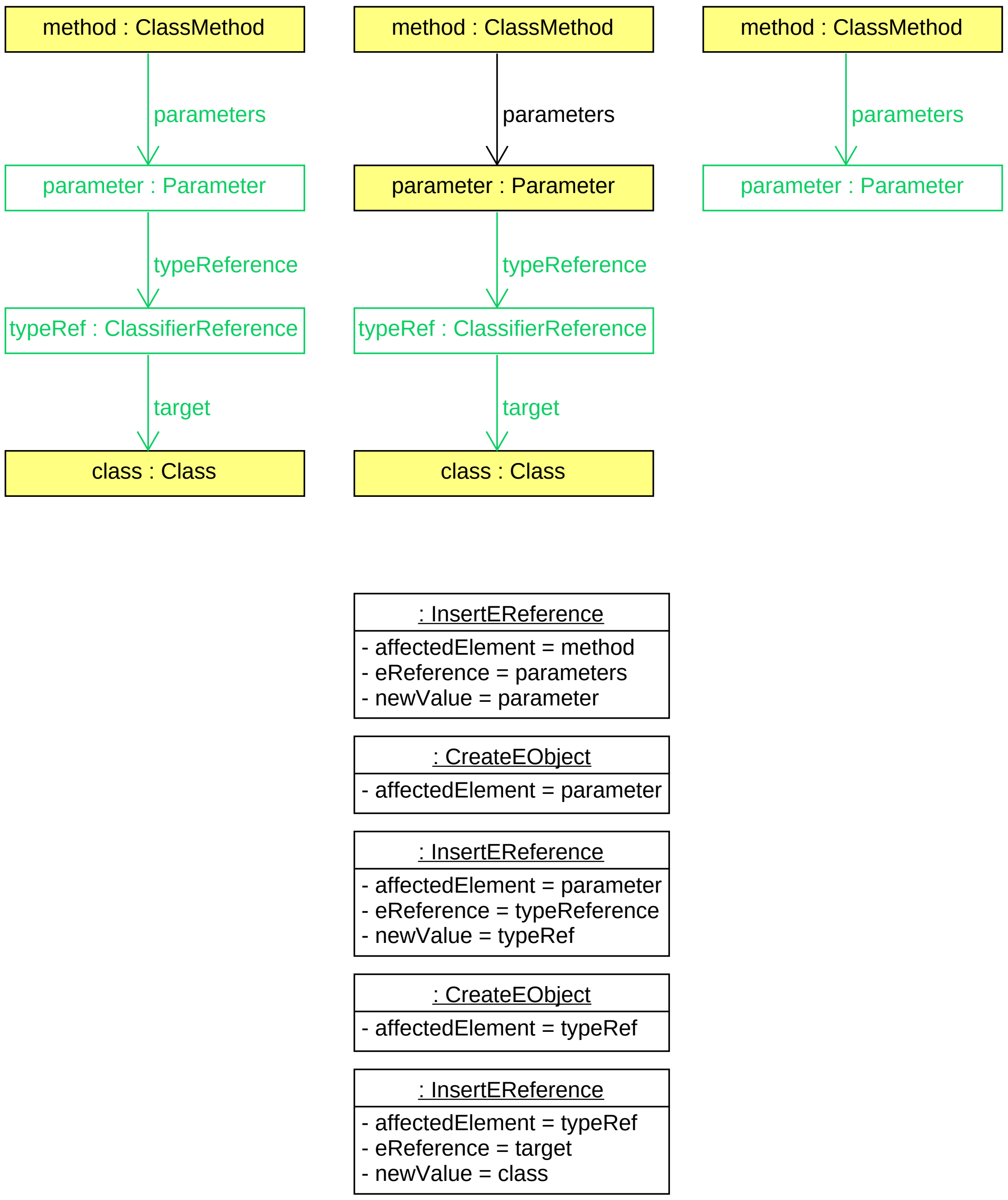

Figure 3.6.: Application example for the containment level heuristic: In the top, three source parts of TGG rules are shown. The left rule fully contains both other rules. Green matching, applied to the change sequence at the bottom of the figure, will invoke all three rules. The dynamic containment level heuristic application algorithm will select the leftmost rule, since it contains the other two.

Algorithm 3.2 Black Pattern Matching: incoming edges

```
function VISITINCOMINGEDGE($incomingEdge : TGGRuleEdge$)
    $srcRuleNode \leftarrow incomingEdge.sourceNode$
    $trgRuleNode \leftarrow incomingEdge.targetNode$
    if VISITED($srcRuleNode$) $\vee$ ISCREATE($srcRuleNode$) then
        return true
    end if
    $trgNodeEObject \leftarrow ruleNodeToEObjectMapStack.get(trgRuleNode)$
    if $incomingEdge.domain = trgRuleNode.domain$ then
        if $incomingEdge.EReference.isContainment$ then
            $ruleNodeToEObjectMapStack.putPush(srcRuleNode, trgRuleNode.eContainer)$
            return VISITNODE($srcRuleNode$)
        else
            $eObjectCandidates \leftarrow$ FINDALLEOBJECTSREFERENCING($trgNodeEObject$,
                                                  $incomingEdge.EReference$)
            return TRYALLMATCHINGCANDIDATESFOR($eObjectCandidates, srcRuleNode$)
        end if
    else                    ▹ $incomingEdge$ is a CORR edge, $srcRuleNode$ is a CORR node
        $correlatedRuleNode \leftarrow$ GETCORRELATEDNODE($trgRuleNode, srcRuleNode$)
        if ISCONTEXT($correlatedRuleNode$) then
            ($corrNodeInstance, correlatedEObject$) $\leftarrow$ FINDCORRINSTANCE(
                                                  $trgNodeEObject$,
                                                  $srcRuleNode$,
                                                  $correlatedRuleNode$)
            if $correlatedEObject$ exists then
                $ruleNodeToEObjectMapStack.putPush(srcRuleNode, corrNodeInstance)$
                $ruleNodeToEObjectMapStack.putPush(correlatedRuleNode, correlatedEObject)$
                return VISITNODE($correlatedRuleNode$)          ▹ Switch to other domain
            end if
        end if
        return true                     ▹ Green node in other domain. Cannot be present.
    end if
    return true
end function
```

Since the containment level heuristic only solves those overlap cases where one rule match completely covers another, two other heuristics are applied for the remaining conflicts. First, since it is a goal of this thesis to preserve consistency completeness, the matches with the highest coverage in the change sequence are chosen, with the intention to minimize the number of leftover EChanges that could have been matched. For all remaining conflicts, the *distance heuristic* is used as defined by Khelladi et al. [25].

The dynamic containment level heuristic is applied as follows: First, rule matches are sorted by the number of EChanges they cover in ascending order. Then, for each rule match r_i,

Algorithm 3.3 Black Pattern Matching: multiple candidate handling

```
function tryAllMatchingCandidatesFor($eObjectCandidates : Set < EObject >$, $ruleNode : TGGRuleNode$)
    if $ruleNodeToEObjectMapStack.containsKey(ruleNode)$ then
        return visitNode($ruleNode$)
    end if
    for all $eObjectCandidate : eObjectCandidates$ do
        $ruleNodeToEObjectMapStack.putPush(ruleNode, eObjectCandidate)$
        if visitNode($ruleNode$) then
            return true
        end if
                    ▷ After each failed descend, reset visit markers and the mapping
        unmarkVisited($ruleNodeToEObjectMapStack.removePopUntil(ruleNode)$)
    end for
    return false
end function
```

each rule match r_j that covers more EChanges than r_i is checked for containment. If r_i is contained in r_j, r_i is discarded.

3.6. Red Matching: Subtractive Changes

In the previous sections, handling *additive* EChanges that *add* something to a source model and how to transport that to the target model has been covered. However, in this concept, handling *subtractive* EChanges that remove something from the model also is considered.

The Vitruvius change metamodel (see section 2.3) also includes EChanges that are both additive and subtractive, the subclasses of *ReplaceSingleValuedFeatureEChange*. These EChanges remove the old value of an attribute or reference of an EObject and add a new value to them. This poses a problem if one wants to map this kind of change to TGG rule applications. In the TGG context, this change is not atomic, so this concept's approach is to split each *ReplaceSingleValuedFeatureEChange* into an additive and a subtractive EChange. In the case of *ReplaceSingleValuedEReference*, these are *InsertEReference* and *RemoveEReference*. This split is done at the very beginning of the change propagation process, before green pattern matching.

In the following, the concept for detecting subtractive changes and repairing the TGG rule application protocol based on an atomic change sequence is described. Without loss of generality, in this section, the assumption that the change sequence contains changes to the source model/graph is made. Handling subtractive changes in TGGs means that we need to find out which previously applied TGG rule matches are broken because of a change to the source model. Subsection 3.6.1 and subsection 3.6.2 describe how that detection is realizable by looking at the TGG rule application protocol and the given atomic change sequence. Considering repair, a common strategy is to invalidate these rule applications and, cascadingly, the rule applications that have context nodes that the now-invalidated rule applications have created. As described in section 2.4, the concept of short-cut rules [15, 14, 13] can be used to avoid this invalidation strategy in some cases, but not in all. Thus, the remaining cases must be dealt with. Invalidating TGG rule applications leaves nodes and edges in the source graph uncovered by matches that potentially could be covered. This problem is tried to be solved by re-creating EChanges from the invalidated rule applications, which is described in subsection 3.6.4, and performing the pattern matching described in section 3.1.

3.6.1. Implicit Detection

The detection of broken matches is split into implicit and explicit detection. Implicit detection is based on the idea that each rule application recorded in the protocol that is missing a node is obviously broken. If one assumes the change detection or recording mechanism that produces the change sequence given by Vitruvius is correct, this means that EChanges typed with *DeleteEObject* are covered with this process.

3.6.2. Explicit Detection

Explicit detection means that each *subtractive* EChange in the given atomic change sequence is looked upon, except *DeleteEObject* and *ReplaceSingleValuedFeatureEChange*. The former is covered in subsection 3.6.1, the latter cannot occur because, as described before, these EChanges are split up into an *InsertEReference* and a *RemoveEReference* EChange.

Different EChanges are handled differently; the procedure is described in the following.

RemoveRootEObject This change can, but does not need to, occur together with a matching *DeleteEObject* EChange, so it needs to be handled separately. From a TGG perspective, it must have the same effects on the rule match in which the affected EObject is identified with a green node. Thus, the rule match where the affected EObject of the RemoveRootEObject EChange occurs as bound to a green node is found and invalidated.

UpdateAttributeEChange In the case of attribute updates, it is not sufficient to only look at subtractive attribute EChanges. This is, because in some TGG formalizations, attributes are not part of the graph but of nodes of a graph. Their values are set via attribute conditions (see subsection 2.5.1). Thus, the setting of an attribute must happen in the same rule that created its affected EObject. That is why all *UpdateAttributeEChanges*, that occur in the same atomic change sequences as the *CreateEObject* EChange that has created the EObject to which the attribute belongs, are ignored in the context of detecting invalid matches. However, those *UpdateAttributeEChanges* where that isn't the case are found by searching for the rule application where the affected EObject of the *UpdateAttributeEChange* is associated with a *create* node. That rule application is invalidated.

RemoveEReference A RemoveEReference EChange deletes a value out of a reference that is contained in an affected EObject. If the reference is many-valued, an index indicates the index in the EList reference. For that kind of EChange, all TGG rule matches where the reference occurs in that constellation are found and invalidated.

UnsetFeature The UnsetFeature EChange stands for unsetting the values of a feature of an affected EObject. This is handled by finding all TGG rule matches where the given feature occurs in combination with the affected EObject and invalidating them.

3.6.3. Fixpoint Iterating Broken Matches

In subsection 3.6.1 and subsection 3.6.2, the concept for determining rule applications that are invalidated by the given model change has been described. Since there might be rule applications that use some of the *create* nodes of these invalidated rule applications as *context* nodes, these rule applications must also be invalidated. Due to the cascading nature of this invalidation, a worklist algorithm that performs fixpoint iteration on the set of invalidated rule matches has been designed (see Algorithm 3.4).

Algorithm 3.4 Fixpoint Iterating Broken Matches

```
function FIXPOINTITERATEBROKENMATCHES($brokenMatches$)
    $worklist \leftarrow$ CREATENODESOF($brokenMatches$)
    while $worklist \neq \emptyset$ do
        $currentCreateNode \leftarrow worklist.remove()$
        $eObject \leftarrow$ GETMODELOBJECT($currentCreateNode$)
        $newBrokenMatches \leftarrow$ INTACTMATCHESCONTAININGASCONTEXT($eObject$)
        $brokenMatches \leftarrow brokenMatches \cup newBrokenMatches$
        $newCreateNodes \leftarrow$ CREATENODESOF($newBrokenMatches$)
        $worklist \leftarrow worklist \cup newCreateNodes$
    end while
end function
```

3.6.4. Repair

As matches are invalidated, model nodes become *unmarked* in the sense that they are not covered by a rule application anymore. This means that the possibility of new rule matches that cover these now unmarked nodes arises. To account for that, this concept recreates EChanges out of those free model nodes. These EChanges form an artificial change sequence that is dealt with as a normal change sequence is, by using section 3.3 and section 3.4.

For recreating EChanges out of invalidated matches, algorithm 3.5 has been designed. For each invalidated match, each green rule node that is still intact, meaning its mapped-to EObject has not been deleted, creates a new *CreateEObject* EChange. If the green rule has no context in the source domain, its mapped-to EObject always is a root EObject, so in that case, an *InsertRootEObject* EChange is also created. Further, for each invalidated match, all green edges that are fully intact in the model, meaning that the source and target EObject are still existing and there exists a reference from source to target that matches the green edge's type, create a new *InsertEReference* EChange.

Algorithm 3.5 Creating EChanges from Broken Matches

```
function createEChangesForBrokenMatch($match$)
    $eChanges \leftarrow \emptyset$
    for all $createNode$ : $match.getIntactCreateNodes()$ do
        $eChanges \leftarrow eChanges \cup$ CreateEObject($createNode$)
        if hasNoIncomingContextOrCreateEdges($ruleNode$) then
            $eChanges \leftarrow eChanges \cup$ InsertRootEObject($createNode$)
        end if
    end for
    for all $createEdge$ : $match.getIntactCreateEdges()$ do
        $eChanges \leftarrow eChanges \cup$ InsertEreference($createEdge$)
    end for
    return $eChanges$
end function
```

4. Implementation

To evaluate and refine the concept and integrate it into the Vitruvius framework, a prototype was implemented. The implementation uses the *eMoflon::IBeX* TGG framework ([38], see section 2.5), which was originally developed as a set of plugins for the *Eclipse* IDE. This made the development process challenging to a certain extent. The prototype was written in Java-21 [22].

In the following, noteworthy aspects of the prototype implementation are described. First, in section 4.1, changes that have been made to Vitruvius to enable propagating sequences of changes at one go are described. Section 4.2 explains details of the realization of *Change Sequence Templates*, their instantiation, matching, and initialization. Then, in section 4.3 details of the pattern matching process that is prescribed by eMoflon::IBeX to a certain extent are shown. Section 4.4 describes how using attribute constraints has been enabled, and section 4.5 concludes by describing challenges with the IBeX framework that have emerged during the implementation of the prototype.

4.1. Changes to Vitruvius

Since the concept requires retrieving a sequence of changes upon which TGG rules are matched, Vitruvius, which previously only supported propagating single `EChanges` one at a time, had to be modified. That was done by adding methods to the `ChangePropagationSpecification` interface that the `TGGChangePropagationSpecification` included in this prototype implements, and changing the propagation functionality in `ChangePropagator`, both being part of the project *Vitruv-Change*.

ChangePropagationSpecification In the ChangePropagationSpecification interface, the two methods seen in Listing 4.1 are added. `doeshandleNonAtomicChange` is added, by default returning `false` to ensure pre-existing ChangePropagationSpecification implementations' behaviour doesn't change and `propagateNonAtomicChange` isn't called if it is not implemented. `propagateNonAtomicChange` is added and called by `ChangePropagator` (see Listing 4.2), to propagate a whole change sequence that is contained in the `VitruviusChange`.

ChangePropagator In `ChangePropagator`, one method is modified and two methods are created to support propagating VitruviusChanges to ChangeSequenceTemplate implementations that implement `propagateNonAtomicChange` (see Listing 4.1). `propagateChanges` now calls propagateNonAtomicChange, too, which is largely based on the existing method `propagateSingleChange`. It is modified to only concern ChangePropagationSpecifications that handle non-atomic changes.
It calls `propagateNonAtomicChangeForChangePropagationSpecification` on each registered ChangePropagationSpecification, which forwards the given `VitruviusChange` to those ChangePropagationSpecifications and collects their effects on their target models. This is the entry point to the prototype.

```
...
def boolean doesHandleNonAtomicChanges() { false }
def void propagateNonAtomicChange(
                        VitruviusChange<EObject> change,
                        EditableCorrespondenceModelView<Correspondence> correspondenceModel,
                        ResourceAccess resourceAccess) {
    // noop
}
```

Listing 4.1: Added methods in *ChangePropagationSpecification.xtend*

```
def private propagateChanges() {
        val result = propagateNonAtomicChange()
      result += sourceChange.transactionalChangeSequence
                    .flatMapFixed[propagateSingleChange(it)]
            ...
}

def private List<PropagatedChange> propagateNonAtomicChange() {
    ...
    val propagationResultChanges = try {
        sourceChange.affectedEObjectsMetamodelDescriptors.flatMap [
            // we only want ChangePropSpecs that handle non-atomic changes
            changePropagationProvider.getChangePropagationSpecifications(it)
                .filter[it.doesHandleNonAtomicChanges()] => [
                    forEach[it.userInteractor = outer.userInteractor]
                ]
        ].toSet.flatMapFixed [
            propagateNonAtomicChangeForChangePropagationSpecification(sourceChange, it)
        ]
    }
    ...
}

def private propagateNonAtomicChangeForChangePropagationSpecification(
    VitruviusChange<EObject> change,
    ChangePropagationSpecification propagationSpecification
) {
    val transitiveChanges = modelRepository.recordChanges [
        propagationSpecification.propagateNonAtomicChange(change,
                modelRepository.correspondenceModel,
                modelRepository)
    ]
    // Store modification information
    changedResources += transitiveChanges.flatMap[it.affectedEObjects]
            .map[eResource]
            .filterNull
    return transitiveChanges
}
```

Listing 4.2: Excerpts of changes to methods in *ChangePropagator.xtend*, enabling change sequence propagation

4.2. Change Sequence Templates

In section 3.2, the concept and structure of *Change Sequence Templates* was described. This section presents noteworthy implementation details of the prototype.

In the prototype implementation, `ChangeSequenceTemplates` and `EChangeWrappers` have different states and roles in the pattern matching process. Those are illustrated in Figure 4.1. The pattern conversion step produces a `ChangeSequenceTemplateSet`. The Change-

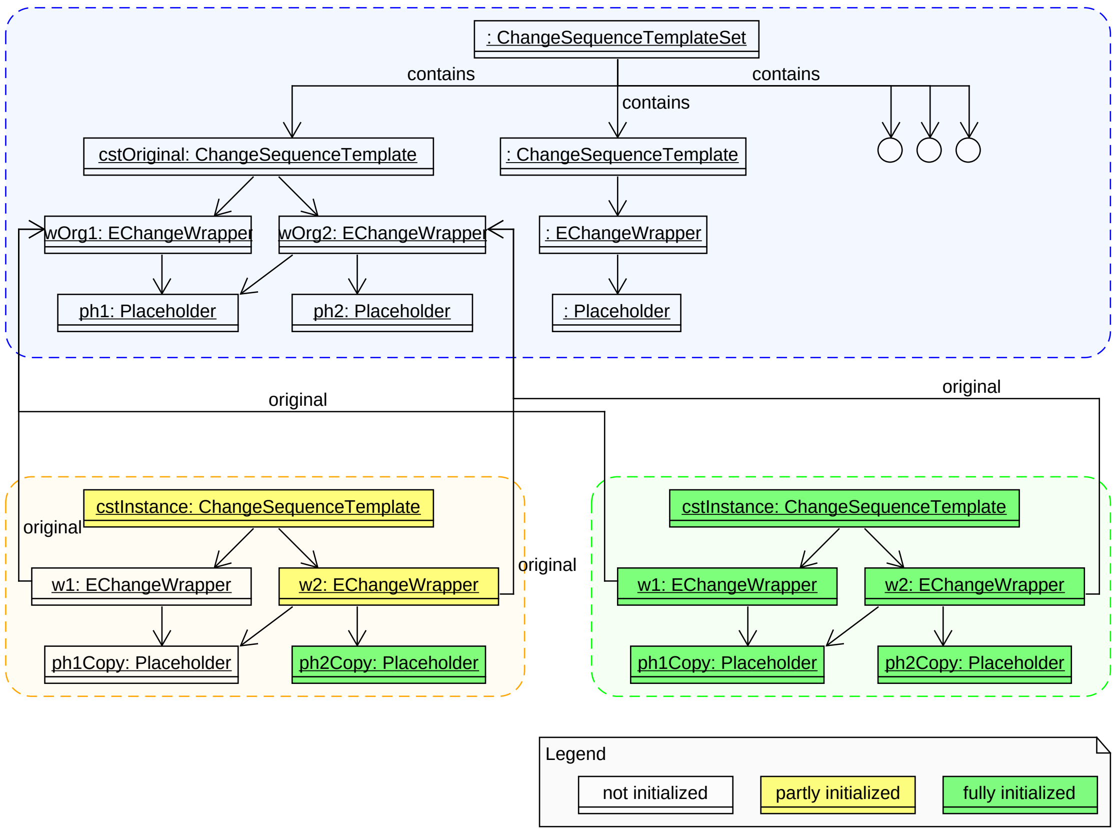

Figure 4.1.: Different states and roles of `ChangeSequenceTemplates`. The originals, which serve as templates for instantiating ChangeSequenceTemplates that serve an instance role, are shown in the blue box, along with the `ChangeSequenceTemplateSet` containing them. The orange box in the bottom left shows an instantiated (note the *original* references) and partly initialized ChangeSequenceTemplate, where one EChangeWrapper is partly initialized and one placeholder is fully initialized. In the green box to the bottom right, the same ChangeSequenceTemplate is shown after it has been fully matched.

SequenceTemplates and their EChangeWrappers contained in that ChangeSequenceTemplateSet have an implicit parent/original role. They do not and will not hold any `EObjects`, they are templates to be copied and thus instantiated.

For instantiation, `getAndInitRelevantChangeSequenceTemplatesByEChange` is called on a ChangeSequenceTemplateSet, returning one or more instances of each ChangeSequenceTemplate that contains an EChangeWrapper that is matchable to an EChange. For each matchable EChangeWrapper, one instance is created and initialized with the EChange, as can be seen in Listing 4.3. To reduce both runtime and complexity, a `HashMap` that maps `EChange EClasses` to the Set of ChangeSequenceTemplates that contain at least one EChangeWrapper of that type is used.

The ChangeSequenceTemplate instances created by `getAndInitRelevantIbexPatternTemplatesByEChange` now each have one EChangeWrapper initialized fully. Because of the placeholder structure, this might indirectly initialize other EChangeWrappers *partially* or *fully*. An EChangeWrapper is fully initialized if all of its `EObjectPlaceholders` are holding an EObject. Thus, at this point, some of the returned ChangeSequenceTemplates are partially or fully initialized (see Listing 4.3). After pattern matching, they are either fully initialized, indicating a successful green match, or discarded.

In the following, the mechanisms of matching and initializing EChangeWrappers are described, followed by an explanation of the copy mechanism.

```java
public Set<ChangeSequenceTemplate> getAndInitRelevantChangeSequenceTemplatesByEChange(
        EChange<EObject> eChange) {
    Set<ChangeSequenceTemplate> partlyInitializedTemplates = new HashSet<>();
    for (ChangeSequenceTemplate changeSequenceTemplate : changeSequenceTemplatesByEChangeType
            .get(eChange.eClass())) {
        changeSequenceTemplate.getEChangeWrappers().stream()
            .filter(eChangeWrapper -> eChangeWrapper.matches(eChange))
            .forEach(eChangeWrapper -> {
                ChangeSequenceTemplate changeSequenceTemplateCopy =
                        changeSequenceTemplate.deepCopy();
                changeSequenceTemplateCopy
                    .getThisInstancesEChangeWrapperFromParent(eChangeWrapper)
                    .initialize(eChange);
                partlyInitializedTemplates.add(changeSequenceTemplateCopy);
            });
    }
    return partlyInitializedTemplates;
}
```

Listing 4.3: Change Sequence Template Instantiation in `ChangeSequenceTemplateSet`

4.2.1. Matching and initialization

The methods for matching and initializing an EChangeWrapper form a pair. They are similarly structured and intended to be called after one another. First, *matches* is called on an EChange, checking equality of the type of the given `EChange` and the EChangeWrappers `EChange` type, and, if present, all `EObjects` contained in placeholders of the EChangeWrapper. This flexibility in matching is needed because EChangeWrappers might be partly initialized by other EChangeWrappers of the same ChangeSequenceTemplates having been initialized before and sharing placeholders. For different kinds of `EChanges`, different subclasses of EChangeWrapper are defined. Those hold different additional fields and placeholders for further `EObjects` that an `EChange` might contain. As subclasses differ in what they hold, matching and initializing an EChangeWrapper must be subclass-specific. To realize that, both methods follow a similar concept: The `EChange` type, meaning its EClass, and the affectedEObject are handled in the abstract `EChangeWrapper` parent class by the methods `initialize` and `matches`. The additional matching checks and initialization are handled in subclasses that implement the abstract methods `initializeExtension` and `extendedDataMatches`, as shown in Listing 4.4.

```java
public boolean matches(EChange<EObject> eChange) {
    return  eChangeTypeAndAffectedEObjectMatches(eChange) && extendedDataMatches(eChange);
}
public void initialize(EChange<EObject> eChange) {
    this.setEChange(eChange);
    this.getAffectedElementPlaceholder().initialize(Util.getAffectedEObjectFromEChange(
    eChange));
    // delegate further initialization to subclasses
    this.isInitialized = true;
}

protected abstract void initializeExtension(EChange<EObject> eChange);
protected abstract boolean extendedDataMatches(EChange<EObject> eChange);
```

Listing 4.4: Matching and initialization in `EChangeWrapper`

4.2.2. Copy mechanism

As mentioned before, ChangeSequenceTemplate *parents* are created by pattern conversion (`IbexPatternToChangeSequenceTemplateConverter`) and instantiated in the process of pattern matching. This raises the need for a copy mechanism that keeps the ChangeSequenceTemplate's EChangeWrapper and placeholder structure. Its implementation is shown in Listing 4.5. First, each EChangeWrapper is copied shallowly, meaning that the placeholders in the new EChangeWrappers remain the same as in their original. The new EChangeWrapper gets a reference to its original by setting the `original` field. A mapping of old to new EChangeWrappers is created. Then, a mapping of old to new `Placeholders` is created and handed to each EChangeWrapper to replace their old placeholders with new ones. Finally, a new ChangeSequenceTemplate is created via a copy constructor that takes the new EChangeWrappers and the map. The map is used by the instance to be able to retrieve EChangeWrapper instances given an `original` EChangeWrapper, as is done in Listing 4.3 by calling `getThisInstancesEChangeWrapperFromParent`.

```java
public ChangeSequenceTemplate deepCopy() {
        Collection<EChangeWrapper> newEChangeWrappers = new LinkedList<>();
        Map<EChangeWrapper, EChangeWrapper> oldToNewEChangeWrapperMap = new HashMap<>();
        for (EChangeWrapper changeWrapper : this.eChangeWrappers) {
            EChangeWrapper newEChangeWrapper = changeWrapper.shallowCopy();
            oldToNewEChangeWrapperMap.put(changeWrapper, newEChangeWrapper);
            newEChangeWrappers.add(newEChangeWrapper);
        }
        Map<EObjectPlaceholder, EObjectPlaceholder> oldToNewPlaceholders = newEChangeWrappers
                .stream()
                .flatMap(eChangeWrapper -> eChangeWrapper.getAllPlaceholders().stream())
                .distinct()
                .collect(Collectors.toMap(
                        eObjectPlaceholder -> eObjectPlaceholder,
                        eObjectPlaceholder -> new EObjectPlaceholder(eObjectPlaceholder
                                .getTggRuleNode())
                ));
        for (EChangeWrapper eChangeWrapper : newEChangeWrappers) {
            eChangeWrapper.replaceAllPlaceholders(oldToNewPlaceholders);
        }
        return new ChangeSequenceTemplate(this.tggRule,
                this.iBeXContextPatternMap, newEChangeWrappers, oldToNewEChangeWrapperMap);;
    }
```

Listing 4.5: Copy mechanism in `ChangeSequenceTemplate`

4.3. Pattern Matching Process

The implementation of the pattern matching process is considerably influenced by the synchronization algorithm and architecture of *eMoflon::IBeX*, which is described in subsection 2.5.2. This section describes the side of the process that concerns the pattern matching, which is driven by being able to correctly implement the method `updateMatches)` in `VitruviusBackwardConversionTGGEngine`.

4.3.1. VitruviusBackwardConversionTGGEngine

This section describes functionality of the class `VitruviusBackwardConversionTGGEngine` that implements the pattern matcher interface of IBeX, called `IBlackInterpreter` and thus serves as a partner to the `SYNC`-extending class `VitruviusTGGChangePropagationIbexEntrypoint`. The interesting parts are described in the following by elucidating noteworthy methods.

initPatterns Implementing the `IContextPatternInterpreter` interface of IBeX, `VitruviusTGGChangePropagationIbexEntrypoint` implements the method `initPatterns`. This method is used to initialize the pattern matchers `VitruviusChangePatternMatcher` and

`VitruviusChangeBrokenMatchMatcher`. Also, pattern conversion is triggered here, using `IbexPatternToChangeSequenceTemplateConverter`.

updateMatches As described in subsection 2.5.2, in `VitruviusBackwardConversionTGGEngine`, the `updateMatches` method, shown in Listing 4.6 is called multiple times by `VitruviusTGGChangePropagationIbexEntrypoint` which extends the IBeX class `SYNC` that, along with further parent classes, defines the synchronization algorithm of IBeX. In the following, the handle `SYNC` is used in descriptions of the process. Between each call and the next, the triple graph might have been changed by `SYNC`. Matches might have been applied or revoked, changing the correspondence graph, the protocol, and the target model. First, so-called *consistency matches* are initialized, if not already present, by loading the *protocol* that stores rule applications from previous runs and transforming them to `VitruviusConsistencyMatches`. That is a class implementing the IBeX interface `ITGGMatch` and is used in the prototype to represent broken matches or intact matches that have been loaded from the protocol file. For unknown reasons, this initialization is not done by IBeX itself. To avoid recalculating matches each time updateMatches is called, `createForwardMatchesIfNotAlreadyPresent` creates green matches one time and saves the result in a class field for further calls. However, since some of these matches might have been already applied, the remaining green matches that have not already been applied and whose green nodes are not already covered by a match having been applied prior to the current run of `updateMatches` are calculated in `getMatchesThatHaventBeenAppliedAndAreStillIntact` (see Listing 4.7). `VitruviusBackwardConversionTGGEngine` knows about matches that have been applied by implementing the IBeX interface `IbexObserver` and observing `MATCHAPPLIED ObservableEvents`. The filtered set of matches is then again thinned out twice: First, each match undergoes context matching, as described in section 3.4. Then, match selection, as described in section 3.5 is applied, before handing the remaining matches to `SYNC`. After having handled the additive matches, broken matches that have not been handled yet are determined using the `VitruviusChangeBrokenMatchMatcher` (see subsection 4.3.2 and reported to `SYNC`. Finally, in `repairUnrepairedBrokenMatches` (see Listing 4.8), broken matches newly invalidated by the `RedInterpreter` used by `SYNC` are tried to be repaired, possibly creating and applying new additive matches (see subsection 3.6.4).

```java
@Override
public void updateMatches() {
    initializePreexistingConsistencyMatchesIfNotAlreadyPresent();

    createForwardMatchesIfNotAlreadyPresent();
    Set<VitruviusBackwardConversionMatch> remainingMatches =
            getMatchesThatHaventBeenAppliedAndAreStillIntact();

    matchContext_flatten_andHandToSYNC(remainingMatches);

    Set<VitruviusConsistencyMatch> brokenMatches = getBrokenMatches();
    brokenMatches.forEach(brokenMatch -> {
        this.iMatchObserver.removeMatch(brokenMatch);
    });

    // Repair matches that are broken and have not been repaired by shortcut rules
    if (ibexOptions.repair.useShortcutRules()) {
        throw new IllegalStateException("Using shortcut rules not supported yet:
            Need to filter out repaired matches from the broken matches...");
    }
    repairUnrepairedBrokenMatches();
}
```

Listing 4.6: Copy mechanism in `ChangeSequenceTemplate`

getMatchesThatHaventBeenAppliedAndAreStillIntact The method `getMatchesThatHaventBeenAppliedAndAreStillIntact`, which is depicted in Listing 4.7, helps ensure that matches given to `SYNC` have not been applied before and are not containing green nodes whose matched-to `EObject` is already covered by another match having been applied before. As mentioned before, matches that have been applied are known because `VitruviusBackwardConversionTGGEngine` implements the IBeX interface `IbexObserver`. From these already applied matches, the `EObjects` covered by green nodes of the TGGRule belonging to the match are collected. Then, the green matches that are to be filtered are sorted out if they were already applied or would cover one of these previously collected, already covered `EObjects`.

```java
private Set<VitruviusBackwardConversionMatch>
    getMatchesThatHaventBeenAppliedAndAreStillIntact() {
        Set<EObject> createdEObjectsAlreadyCovered = this.matchesThatHaveBeenApplied.stream()
                .map(match -> (VitruviusBackwardConversionMatch) match)
                .flatMap(match -> match.getEObjectsCreatedByThisMatch().stream())
                .collect(Collectors.toSet());

        return this.matchesFound.stream()
                .filter(match -> !this.matchesThatHaveBeenApplied.contains(match))
                .filter(match -> match.getEObjectsThisMatchWouldCreate().stream()
                        .noneMatch(createdEObjectsAlreadyCovered::contains))
                .collect(Collectors.toSet());
    }
```

Listing 4.7: Match filtering in `VitruviusBackwardConversionTGGEngine`

repairUnrepairedBrokenMatches The method `repairUnrepairedBrokenMatches` (see Listing 4.8) tries to repair broken matches detected by `VitruviusChangeBrokenmatchMatcher` and invalidated by `IbexRedInterpreter`. It does so by trying to repair broken matches using the `UnrepairedBrokenMatchOldChangesRetriever` to generate a new change sequence from those green nodes and edges of the broken matches that are still mapped to `EObjects` (see Figure 4.2 for an example). This new change sequence is extended by adding all `EChange` that were left unmatched by the pattern matching of the “main” change sequence, the one that had started the change propagation process, that have not already been applied. This enables matches overspanning both change sequences to be possible. The unmatched `EChanges` from the “main” change sequence are appended at the *end* of the new change sequence to not distort the application of the distance heuristic (see section 3.5 and section 2.6). In the example in Figure 4.2, the `EChange` representing the creation of Y is in the main change sequence, and all `EObjects` that are referenced by green nodes (and their references) in the bottom graph get `EChanges` created in the new change sequence. The resulting change sequence is given to a new `VitruviusChangePatternMatcher`, which calculates new matches. These new matches are stored globally for `updateMatches` to hand them to `SYNC`. Finally, the new matches are context-matched, coverage-flattened, and handed to `SYNC`, in case the current call to `updateMatches` and `repairUnrepairedBrokenMatches` was the last.

4.3.2. Broken matches detection

The `VitruviusChangeBrokenMatchMatcher` implements what is described in subsection 3.6.1 and subsection 3.6.2. However, the prototype has one limitation to that: `EList` indices are not stored in the protocol. This means that the detection of RemoveEReference is not complete, because in case an element occurs twice in the list, the rule application match where that list insertion was marked with create nodes and edges is ambiguous. Because of that, removing elements in lists that occur there more than once is not supported by the prototype.

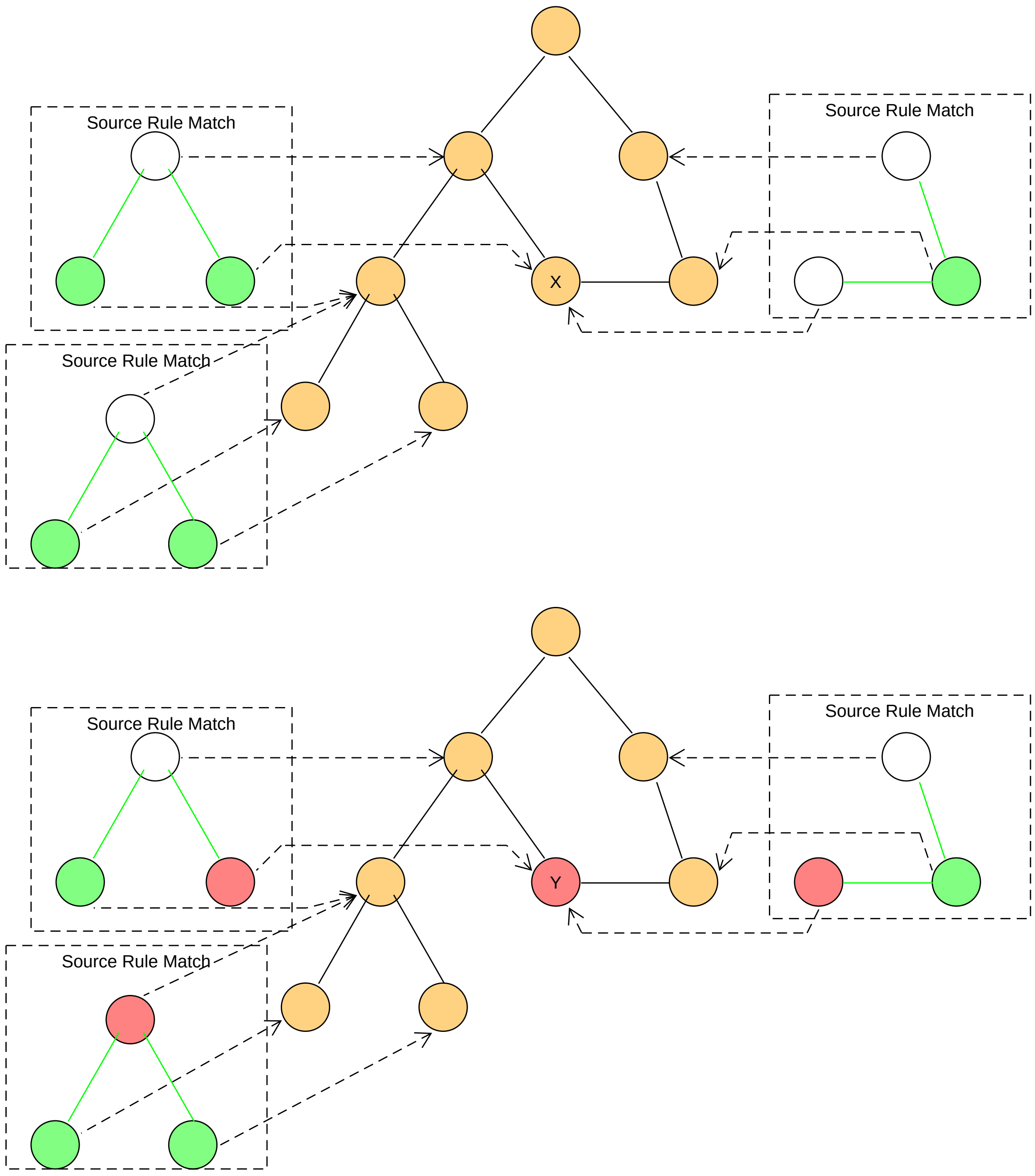

Figure 4.2.: TGG rule matches becoming broken after X is replaced by Y. All three rule matches are broken, and nodes remaining green in the bottom graph are now “free” to be covered by another rule match.

```java
private void repairUnrepairedBrokenMatches() {
    Set<VitruviusConsistencyMatch> unrepairedAndUntriedBrokenMatches = this.
    vitruviusTGGIbexRedInterpreter.getRevokedRuleMatches().stream()
            .map(match -> (VitruviusConsistencyMatch) match)
            .filter(match -> !matchesThatHaveBeenTriedToRepair.contains(match))
            .collect(Collectors.toSet());
    if (unrepairedAndUntriedBrokenMatches.isEmpty()) {
        return; // no point
    }

    List<EChange<EObject>> newChangeSequence = new UnrepairedBrokenMatchOldChangesRetriever(
            this.observedOperationalStrategy.getResourceHandler(),
            this.ibexOptions.tgg.flattenedTGG().getRules().stream()
                    .filter(tggRule -> !tggRule.isAbstract())
                    .collect(Collectors.toSet()),
            unrepairedAndUntriedBrokenMatches, propagationDirection
            ).createNewChangeSequence();
    newChangeSequence.addAll(vitruviusChangePatternMatcher.getUnmatchedEChanges());
    VitruviusChangePatternMatcher newVitruviusChangePatternMatcher =
        new VitruviusChangePatternMatcher(
                VitruviusChangeFactory.getInstance()
                        .createTransactionalChange(newChangeSequence),
                changeSequenceTemplateSet
    );
    Set<VitruviusBackwardConversionMatch> newMatches =
            newVitruviusChangePatternMatcher.getAdditiveMatches(propagationDirection);
    matchesThatHaveBeenTriedToRepair.addAll(unrepairedAndUntriedBrokenMatches);
    this.matchesFound.addAll(newMatches);
    matchContext_flatten_andHandToSYNC(newMatches);
}
```

Listing 4.8: repair handling in `VitruviusBackwardConversionTGGEngine`

4.4. Enabling Attribute Constraints

As described in subsection 2.5.1, eMoflon::IBeX supports defining attribute constraints to specify relationships between attributes of corresponding `EObjects`. Because IBeX isn't designed as a library but as an Eclipse plugin, this was realized via class loading and the *Java Reflection API*, which is shown in Listing 4.9. The user must have defined custom attribute constraints in the default factory for that, which is `UserDefinedRuntimeTGGAttrConstraintFactory`. That Class, although auto-generated in the process of an eMoflon build, always is contained in the same package, namely `org.emoflon.ibex.tgg.operational.csp.constraints.factories.«projectName»`. It is loaded and then instantiated and handed to IBeX.

```java
private void tryToFindAndAddUserDefinedAttributeConstraints(IbexOptions ibexOptions) {
    try {
        //class loader should have access to this CL's classes as well as the ibex
project
        Class userDefinedConstraintFactoryClass = new SimpleNameSupportingURLClassLoader(
                new URL[]{new File(ibexProjectPath, "/bin").toURI().toURL()},
                this.getClass().getClassLoader())
                .loadClass("org.emoflon.ibex.tgg.operational.csp.constraints.factories."
                        + ibexOptions.project.name().toLowerCase()
                        + ".UserDefinedRuntimeTGGAttrConstraintFactory");
        ibexOptions.csp.userDefinedConstraints((RuntimeTGGAttrConstraintFactory)
                userDefinedConstraintFactoryClass.getConstructor().newInstance());
    } catch (MalformedURLException | InvocationTargetException | InstantiationException |
            IllegalAccessException | NoSuchMethodException e) {
        logger.warn("Couldn't load UserDefinedRuntimeTGGAttrConstraintFactory");
    }
}

```

Listing 4.9: Enabling Attribute Constraints in `VitruviusTGGChangePropagationRegistrationHelper`

4.5. Challenges with the IBeX Framework

During the implementation of the prototype in eMoflon, a number of issues with the IBeX framework occurred, some of which could be dealt with, while others could not. Especially the proxy issue, which had an impact on the evaluation results (see subsection 5.3.1), artificially restricts the capabilities of the prototype, preventing rule applications that otherwise would be functioning.

4.5.1. Solved Issues

Some issues with the IBeX framework could be solved by overriding classes and methods from the framework. Some of these issues are bugs, others are a lacking of access to data required by the prototype implementation.

SmartEMF On their website, the eMoflon developers describe *SmartEMF* as a "High-performance EMF reimplementation complying with EMF interfaces" [9]. However, parts of the implementation do not comply with the EMF interfaces, which produced problems while testing the prototype. One of those is `org.emoflon.smartemf.persistence.AdapterList`.

org.emoflon.smartemf.persistence.AdapterList is a SmartEMF implementation of an EMF `EList<Adapter>`. Because its internal representation is a `LinkedHashSet`, which doesnt implement `List` functions, methods like `get(int index)`, `add(int index, Adapter element)` or `indexOf(Object o)`, which have to be implemented to comply with the `EList` interface, miss data to return something meaningful. Thus, the developers of SmartEMF chose to throw an UnsupportedOperationException wherever these methods are called. In preparing the evaluation, such a call occurred while loading model elements. This issue was fixed in the tools.vitruv.dsls.tgg.emoflonintegration.ibex.smartEmfFix package by extending AdapterList with the AdapterListFixed class, overriding the aforementioned indexOf and add methods. indexOf always returns 0, and add is implemented as a noop. This fixed the problem but should be kept in mind in case any persistence issue arises. The AdapterListFixed was "injected" by also overriding the SmartEMFResource and the SmartEMFResourceFactoryImpl classes and registering the latter as the default ResourceFactory for the resourceSet of VitruviusBackwardConversionTGGEngine.

org.emoflon.ibex.tgg.operational.repair.strategies.AttributeRepairStrategy is an IBeX class that enforces attribute constraints on matches. However, it also tries to enforce these on matches that are not fully matched, meaning that some nodes are not mapped to EObjects, which then results in a NullPointerException. This has been fixed by extending the AttributeRepairStrategy with FixedAttributeRepairStrategy in the tools.vitruv.dsls.tgg.emoflonintegration.ibex package. There, the repair method was overridden and a null check added. Similar to the AdapterList issue, the FlexibleSeqRepair class, extending org.emoflon.ibex.tgg.operational.repair.SeqRepair, was created to inject the FixedAttributeRepairStrategy.

paranoid modifications In the VitruviusBackwardConversionTGGEngine, which implements the IBlackInterpreter interface, the getProperties method can be overridden to, among others, set the needs_paranoid_modifications property. While conducting the evaluation, it turned out that both states are needed. For deleting changes, needs_paranoid_modifications being set to false leads to an UnsupportedOperationException being thrown in the UML EMF implementation.
For additive changes, needs_paranoid_modifications being set to true makes the serialization faulty, losing some references that look like they are serialized but aren't deserialized. This issue was again fixed by setting needs_paranoid_modifications to false by default and extending an IBeX class, in this case IbexRedInterpreter. The extending class, VitruviusTGGIbexRedInterpreter, overrides the revoke(...) method where it sets needs_paranoid_modifications temporarily to true, as can be seen in Listing 4.10.

```
@Override
public void revoke(Set<EObject> nodesToRevoke, Set<EMFEdge> edgesToRevoke) {
    patternMatcher.setNeeds_paranoid_modificiations(true);
    super.revoke(nodesToRevoke, edgesToRevoke);
    patternMatcher.setNeeds_paranoid_modificiations(false);
}
```

Listing 4.10: Source Model Serialization

4.5.2. Unsolved Issues

Unfortunately, some issues with eMoflon::IBeX could not be solved. As mentioned before, this distorts the evaluation results, which is also discussed in the evaluation chapter (chapter 5). In the following, these issues are described, as well as the reasons why they could not be solved.

Unresolved proxies The proxy problem appeared in the process of the evaluation. Some `TGGRules` contained nodes and/or references whose `type` attribute, which references the `EClass` or `EReference` that a node or an edge represents, was a *proxy* (see section 2.1), indicating that this metamodel element could not be loaded or has been unloaded. These rules could be translated to `ChangeSequenceTemplates` by the prototype but not be matched to a `VitruviusChange`, because the proxy `EClasses` and `EReferences` could not be matched successfully to the `EObjects`' `EClasses` and the `EReferences` held by the `EChanges`. They, however, were no proxies, which indicates that the problem does not lie with the model or its serialization being faulty but rather with the IBeX framework. All attempts to solve this problem failed, and since at the present time development of the IBeX framework is discontinued, a solution is likely only to be found by either forking the framework or moving to another, e.g., *eMoflon::neo* [37].

Correspondence Model Serialization Another issue that also occurred while conducting the evaluation concerned IBeX's serialization of the correspondence model. This issue is illustrated with an example, as can be seen in Listing 4.11 and Listing 4.12. In a situation where a many-valued reference has two or more values, a deletion of any but the last value results in the correspondence model serialization being corrupted. That is because, as can be seen in Listing 4.12, correspondence entries reference source and target entries by pointing at indexes in the respective model serialization files. Deleting the first `members` entry named "constructer" in Listing 4.11 leads to the `source` entry in Listing 4.12 now pointing to the formerly second, but now first `members` entry in Listing 4.11, that is called "methed". In this case, IBeX won't even load the correspondence model, because the type checking of `JavaConstructorToOperation` fails, since it expects a `Constructor` as source but gets a `ClassMethod`. If the first member were also a member, this corruption would be silent. Like with the unresolved proxies problem, this might be solved by using another framework or implementing an alternative serialization solution that does not relate objects by position but, e.g., by ID.

```xml
<classifiers xsi:type="classifiers:Class" name="kless">
    <members xsi:type="members:Constructor" name="constructer">
       <parameters xsi:type="parameters:OrdinaryParameter" name="parrametr"/>
    </members>
    <members xsi:type="members:ClassMethod" name="methed">
       <parameters xsi:type="parameters:OrdinaryParameter"/>
    </members>
    <members xsi:type="members:Field" name="myField"/>
</classifiers>
```

Listing 4.11: Source Model Serialization

```xml
<Java2Uml:JavaConstructorToOperation>
    <target xsi:type="uml:Operation" href="uml_instance.model#/0/@packagedElement.0/
   @ownedOperation.0"/>
    <source xsi:type="members:Constructor" href="java_instance.model#//@compilationUnits.0/
   @classifiers.0/@members.0"/>
</Java2Uml:JavaConstructorToOperation>
```

Listing 4.12: Correspondence Model Serialization

5. Evaluation

The purpose of the evaluation of the implemented prototype is to empirically assess the capabilities of the developed approach to pattern matching of TGG rules on atomic model changes.

Therefore, one concern is to determine the automatization degree in terms of consistency completeness, i.e., the expressiveness for consistency conditions, the rule representation complexity (the avoidance of data loss) and performance of TGG usage on available Vitruvius evaluation cases like those used in the article describing the Vitruvius approach [28] and based on that compare them with preexisting consistency preservation languages. To that end, the evaluation cases were extended, translating existing rules to TGG rules wherever possible.

A second concern, that of scalability, was evaluated by randomly generating models from the TGG rules in the evaluation case *HospitalToAdministration*, which is from the emoflon::IBeX tutorial [8, 10]. The idea is to generate models of various sizes and measure the asymptotical runtime performance of the prototype.

5.1. GQM Plan

The evaluation is structured by following a Goal Question Metric (GQM) plan, as proposed by Basili et al. [2], to enable transparent methodical verifiability and to explicitly consider construct validity.

G1 Ensuring correctness of the consistency preservation process

Q1.1 Does the approach identify complex changes correctly?

M1.1.1 Number of change sequences that are incorrectly matched

G2 Preserving consistency completeness

Q2.1 Can realistic consistency relations be represented by the TGG approach?

M2.1.1 Number of rules that can be represented by TGG rules.

G3 Preserving or improving performance of consistency preservation

Q3.1 How does the approach perform in comparison to existing approaches?

M3.1.1 Runtime trend of consistency preservation algorithm on change sequences in dependence of input nodes and change sequence length.

M3.1.2 Runtime trend of different subroutines of the consistency preservation algorithm on change sequences in dependence of input nodes and change sequence length.

In section 5.2, the realization of the GQM plan are explained in full detail. Section 5.3 presents and discusses the results of the evaluation according to the method described in section 5.2.

5.2. Method

In this section, the general preconditions for conducting the evaluation are described. This includes the used evaluation cases and the procedure. Then, each goal described in the GQM Plan (see section 5.1) and their respective setup constellations are explained.

Evaluation Cases In the preparation of this evaluation, two evaluation cases were chosen for the accomplishment of the three goals to depend on:

- **G1**, **G2**: *JavaToUml* [36, 4]
- **G3**: *HospitalToAdministration* [8, 10]

In the case of *JavaToUml*, existing consistency preservation rules, written in the reactions language [29], were attempted to translate into TGG rules by hand.

5.2.1. G1 - Ensuring correctness of the consistency preservation process

Ensuring correctness of the consistency preservation process is the first goal of the evaluation of the prototype. The TGG rules written for the two evaluation cases form the basis for being able to reach that goal. The following question is answered:

Q1.1 Does the approach identify complex changes correctly?
To answer that question, we define a number of change sequences so that each TGG rule is covered at least once. We further define which matches are expected for each change sequence. Based on that, the number of change sequences whose matches deviates from the expectation, and are thus *incorrectly matched*, is determined. Further, that same number of incorrectly matched patterns under the fiction of not having the proxy problem described in subsection 4.5.2 is determined. Due to the fictional nature, this is done by comparing the change sequences that are incorrectly matched due to the proxy problem to other change sequences that don't fall under the proxy problem and assuming that they would behave in the same way. For each of those, a "partner rule" is determined to make this metric more traceable and replicable.

Since the outcome of that metric and its proxy-problem ignoring counterpart depends on how change sequences are chosen, their choice is oriented towards the evaluation case.

5.2.2. G2 - Preserving consistency completeness

To evaluate to what extend the goal of preserving consistency completeness is reached, the following question is answered:

Q2.1 Can realistic consistency relations be represented by the TGG approach?
The choice of using the evaluation case *JavaToUml* for the evaluation roots in this question. Both Java and UML are non-artificial metamodels which have been modeled in the Eclipse Modeling Framewok. Since there are overlaps in the concepts that both metamodels describe, consistency relations between both metamodels can be deemed realistic and have also been modeled in the *Reactions* language [29] prior to this thesis. To answer the question, that *consistency preservation rules* (CPRs), represented by reactions, are looked upon. The effects of each reaction are tried to be modeled with TGG rules that are processable in the prototype. Then, each rule is categorized into one of the following categories:

- *Full preservation*: The consistency preservation rule is equivalently implemented by both the reaction and the TGG rule(s).
- *Partial preservation*: The consistency preservation rule expressed by the reaction can only be implemented partially by TGG rules. Some aspects are implementable and processable, some are not.
- *No preservation* The consistency preservation rule can not be expressed with TGG rules or not be processed by the prototype

For each category, the number of CPRs that fall into that category is determined. For partial or no preservation, reasons are given.

5.2.3. G3 - Preserving or improving performance of consistency preservation

Having set the goal of preserving or improving performance of consistency preservation with the concept, the following question is to be answered:

Q3.1 How does the approach perform in comparison to existing approaches?
The question of performance and scalability rises as using preexistent information on the actual changes to generate matches is an approach in which the search space is smaller and thus, it can be hypothesized that less computing time is needed to find rule matches and propagate them to the target model. However, since the concept and prototype include a selection of pattern matches (see section 3.5), this must be taken into account when comparing the approach to another pattern matcher that does not perform this. The other pattern matcher is the *high-performance pattern matcher* HiPE [17] included in the

eMoflon::IBeX framework. For easier referencing, the approach in this thesis is called VitruvTGG in the context of evaluation. The following time measurements are taken:

- VitruvTGG:
  - Main green forward matching time: The time spent for the green matching of the change sequence against the ChangeSequenceTemplates
  - Total context matching time: The cumulated time spent on trying to match context for green matches that have not yet been applied.
  - Total coverage flattening time: The cumulated time spent on *coverage flattening*, i.e. ensuring that each change is covered by at most one match.
  - Total change propagation time: The total runtime of the change propagation
- HiPE: Total change propagation time: The total runtime of the change propagation

Experimental Setup The experimental setup is designed in the following way: Models, and thus change sequences, of different sizes are generated randomly with the eMoflon::IBeX Modelgen application. This application generates a “random” triple graph based on a TGG ruleset. Modelgen has limitations that restrict the choice of model size measure. While it is possible to specify a maximum amount of graph elements this cannot be the only constraint since Modelgen only applies one rule if only that constraint is given. To nudge Modelgen to more randomness, the max count of rules is also included, calculated as follows: $maxApplicationsPerRule = graphElementSize/numberOfRules$. Although a rule application always creates at least one graph element, These further constraints have repercussions on the resulting graph element size, because situations might occur where some rules have reached their max count and all others are not applicable to the current situation anymore. The procedure is as follows: For $graphElementSize = 2^3, 2^4, \dots, 2^{11}$, the following steps are taken with both the HiPE and the VitruvTGG pattern matcher:

1. Generate triple graph with source element size *graphElementSize* and the aforementioned additional constraints.
2. Delete correspondences, protocol and target graph.
3. Generate a change sequence from $\emptyset$ to *model* using Vitruvius classes that use EMF Compare [7].
4. Start change propagation and measure the times described above.
5. Repeat steps 1 to 4 with this parametrization 20 times.
6. calculate the median and store the results.

That procedure is automated using *JUnit 5* [23]. Performance measurements are executed on a machine with the following specifications:

- Processor: Intel Core i9-8950HK
- Random Access Memory: 32.0 GB, 2667 MHz

- Operating System: Windows 10 Pro

5.3. Results

In this section, the results of the evaluation that was performed following the method in section 5.2 are presented.

5.3.1. G1 - Ensuring correctness of the consistency preservation process

In Table 5.1, the results of evaluating the prototype on *JavaToUML* with metric *M1.1.1* can be seen. It is noticeable, that while with 57.78%, the number of correctly matched patterns can be considerd relatively low, the number of correctly matched patterns if there was no proxy problem is relatively high (95.56%). That can be interpreted as a good sign for the correctness of the concept but a drawback for the prototype or the framework that introduces the proxy problem.

	matching categories			
	total	incorrectly matched	correctly matched	correctly matched if there was no proxy problem
Number of change sequences	45	19	26	43
Proportion of total	100%	42.22%	57.78%	95.56%

Table 5.1.: Results of metric M1.1.1 on evaluation case JavaToUml

5.3.2. G2 - Preserving consistency completeness

In Table 5.2, the results of evaluating the prototype on *JavaToUML* with metric *M2.1.1* can be seen. With 93.33%, consistency completeness can be interpreted as high, considering that the baseline is a turing-complete imperative language while the protoype's imperativity is limited to attribute conditions.

	representability			
	total	full	partial	None
Number of consistency preservation rules	45	42	2	1
Proportion of total	100%	93.33%	4.44%	2.22%

Table 5.2.: Results of metric M2.1.1 on evaluation case JavaToUml

```
reaction JavaCompilationUnitInsertedInPackage {
  after element java::CompilationUnit inserted in java::Package[compilationUnits]
  call {
    rootElementPreSetup(newValue)
    newValue.classifiers.forEach[insertUmlClassifierIntoPackage(it, affectedEObject)]
  }
}
routine rootElementPreSetup(EObject alreadyPersistedEObject) {
  update {
    detectOrCreateUmlModel(alreadyPersistedEObject)
    registerPredefinedUmlPrimitiveTypes(alreadyPersistedEObject)
  }
}
routine registerPredefinedUmlPrimitiveTypes(EObject alreadyPersistedEObject) {
  update {
    getNotRegisteredPrimitiveTypesWithUnifiedNames([ sourceElement, expectedType, tag |
      getCorrespondingElement(sourceElement, expectedType, null, tag, false)
    ], alreadyPersistedEObject.eResource.resourceSet).forEach [
      addCorrespondenceBetween(UMLPackage.Literals.PRIMITIVE_TYPE, it.key, it.value)
    ]
  }
}
```

Listing 5.1: Reaction JavaCompilationUnitInsertedInPackage

The cases where the consistency preservation rules defined in the *Reactions* langugage could not be fully represented and the reasons for that are discussed in the following.

Reactions that could only be partially reppresented with TGG rules include `JavaCompilationUnitInsertedInPackage` and `JavaClassifierRemoved`. `JavaCompilationUnitInsertedInPackage`, as can be seen in Listing 5.1 includes adding correspondences between the metamodel element `UMLPackage.Literals.PRIMITIVE_TYPE` and model elements. This cannot be mapped with TGGs.

The reaction JavaClassifierRemoved could only be represented partly, because in the TGG implementation used in the approach, deleting a model element always means invalidating the rule that has created it. In JavaClassifierRemoved, as can be seen in Listing 5.2, if a $Classifier$ is removed, the $CompilationUnit$ it belongs to is automatically removed, too. However, this relation is not symmetric in a sense that $Classifier$ and $CompilationUnit$ are always required to be created together, since two reactions are responsible for that, `JavaCompilationUnitInsertedInPackage` and `JavaClassifierInserted`. Representing this in TGGs either means representing these two reactions with one TGG rule for each reaction, as can be seen in Figure 5.1, or representing both reaction with a single TGG rule. In the former case, the option to create a $Classifier$ independent of its $CompilationUnit$ gets lost, thus those two would only be partly represented. In the latter case, which is the one that was chosen, `JavaClassifierRemoved` is only represented partly, as only the $Classifier$ would be removed by revoking the respective additive rule, and the $CompilationUnit$ would survive.

```
reaction JavaClassifierRemoved {
  after element java::ConcreteClassifier removed from java::CompilationUnit[classifiers]
  call cleanUpJavaCompilationUnit(affectedEObject)
}
routine cleanUpJavaCompilationUnit(java::CompilationUnit jCompUnit) {
  update {
    removeObject(jCompUnit)
  }
}
```

Listing 5.2: Reaction JavaCompilationUnitInsertedInPackage

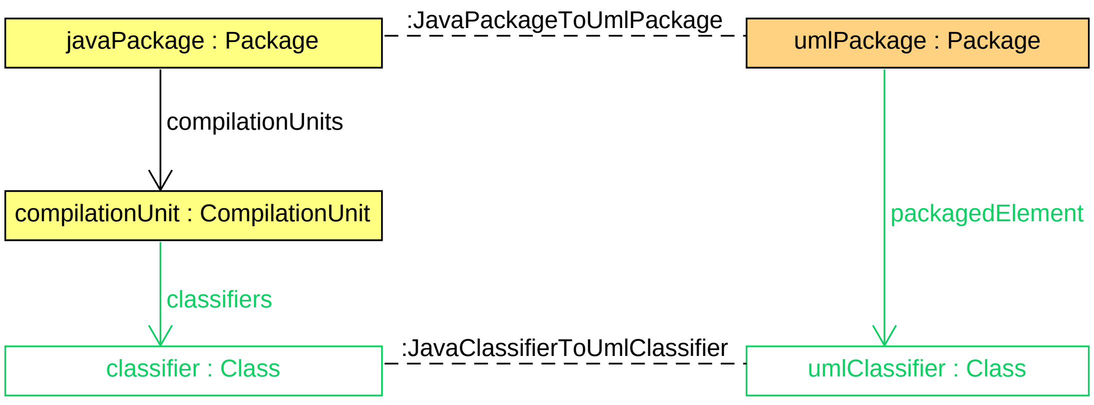

Figure 5.1.: TGG rule ClassToUmlClass

5.3.3. G3 - Preserving or improving performance of consistency preservation

In Figure 5.2, the results of evaluating the prototype on *HospitalToAdministration*, a case from the emoflon::IBeX tutorial [8, 10], using metric *M3.1.1* can be seen. VitruvTGG performs slightly better or equal to HiPE up to a model size of 128 model elements (VitruvTGG: 116ms, HiPE: 115ms). From there on, VitruvTGG performance diverges strongly from HiPE's performance. To find out the reason for that, sub-measurements are included in Figure 5.2. Looking at these, it stands out that coverage flattening takes up most of the total time of VitruvTGG, from 216 model elements upwards it even surpasses the total execution time of HiPE, while the more matching-relevant measurements *main green matching* and *context matching* stay faster than HiPE, although from 4096 model elements upwards, in tha case of *main green matching*, that can be expected to change as well, but not drastically, regarding the course of the curves. Since *coverage flattening*, while being useful to find the *intended* rule matches (as discussed in [25]), is not necessary for the *correctness* of the matching, it is of interest to look at the performance of VitruvTGG if coverage flattening is omitted. That is simulated by subtracting the coverage flattening time from the total change propagation time and shown in Figure 5.3. There it can be seen what has been suspected:

While VitruvTGG performs better until a model size of 256 or 512 elements, performance diverges from there on upwards, although not as steeply as before.

To summarize, it can be said that **Q3.1** has to be answered in a differentiated way: In comparison to HiPE, the approach performs better or equally up to a problem size of 128 and worse from there on upwards. So, whether **G3** has been reached depends on what problem sizes are regarded: Performance is improved or preserved and **G3** is reached for problem sizes smaller as 128 changes in a sequence, for larger sequences, it is not and **G3** is not reached.

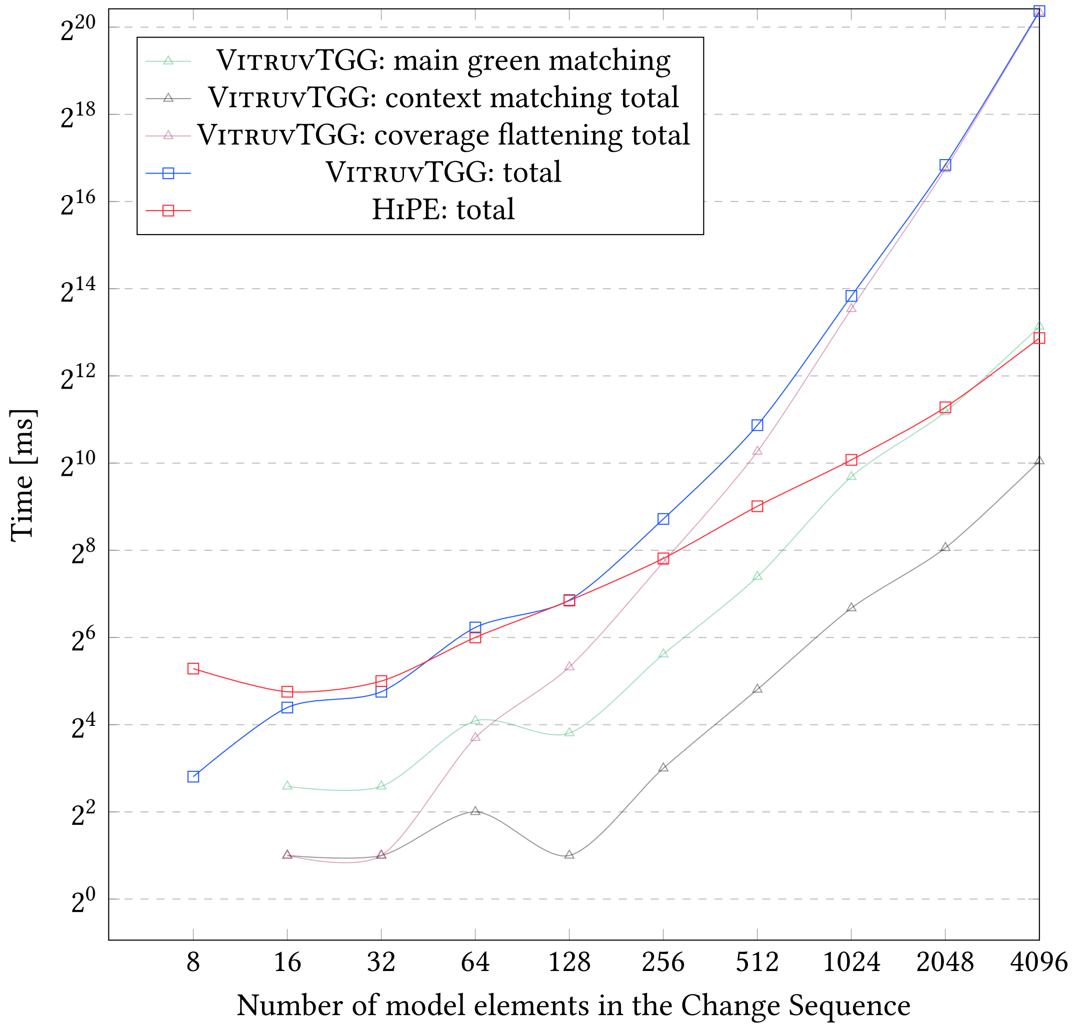

Figure 5.2.: Runtime trend of VitruvTGG and HiPE, median of 20 runs per measurement

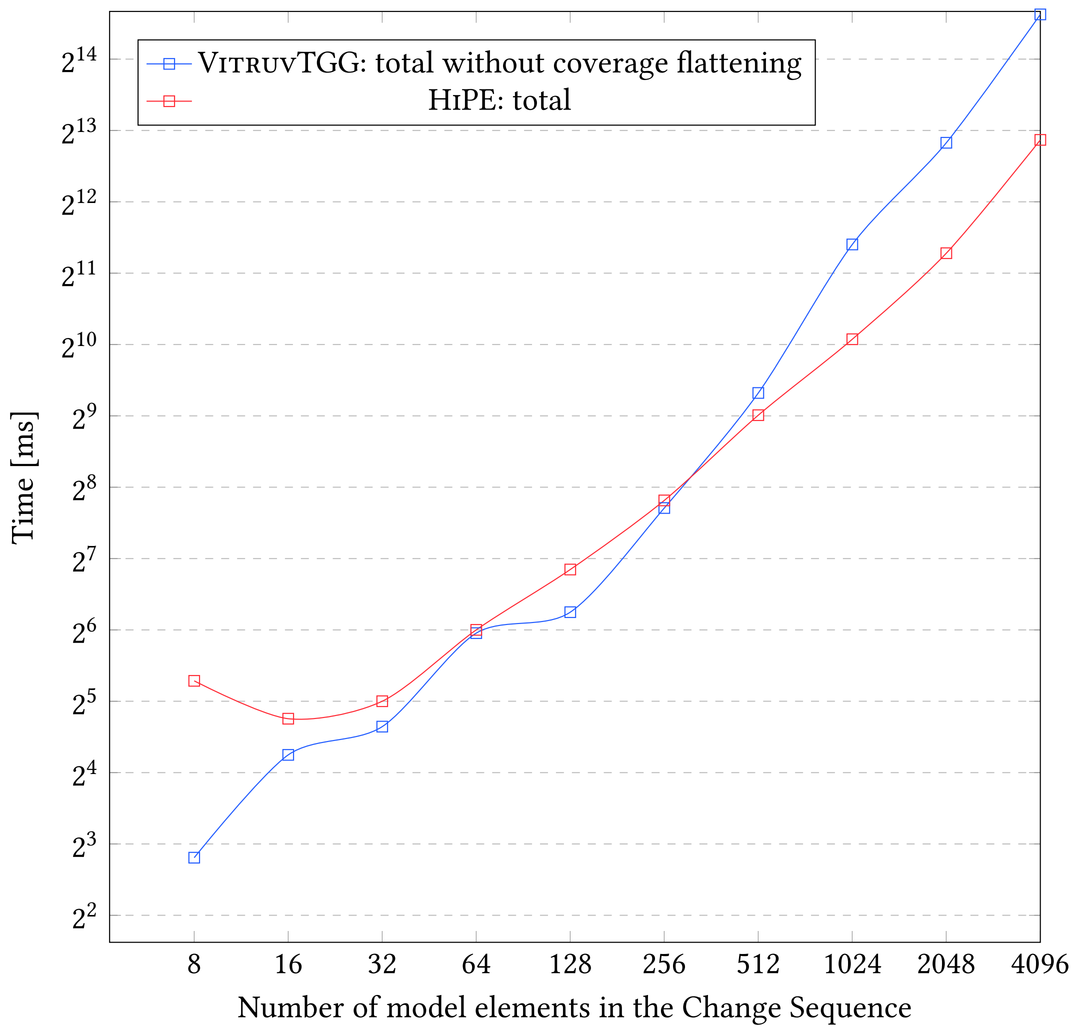

Figure 5.3.: Runtime trend of VitruvTGG without coverage flattening and HiPE, median of 20 runs per measurement

6. Related Work

Other approaches to consistency preservation via model transformations are presented in this chapter.

Domain Specific Languages (*DSLs*) that can be used for model synchronization can be subdivided into unidirectional and bidirectional, as well as into imperative and declarative approaches. TGGs are bidirectional and declarative, which comes with drawbacks and advantages in comparison to other approaches that have been applied or are applicable in the context of V-SUMs. The Vitruvius project currently uses two languages that are used for consistency preservation between models: the *Reactions* Language and the *Commonalities* Language.

The *Reactions* language [29] is an imperative and unidirectional language that allows for the definition of reactions that are executed on models when a change in one model that breaks consistency with another model occurs. Its advantage lies in its expressiveness as a "general-purpose programming language"[29]. In comparison to bidirectional approaches to model synchronization, unidirectional approaches have the disadvantage that transformations have to be written for both directions, which makes implementing them more time-consuming and possibly more error-prone. Further, using a graph-based language like TGGs, complex relationships are visualizable, which can prove user-friendly for methodologists that develop CPRs.

The *Commonalities* approach [27] is motivated by the idea of making common concepts shared between two metamodels explicit and thus making this redundant information visible. These shared concepts are called *Commonalities*, and they are the core entities of a *concept metamodel* whose instances are what the user specifies by using the Commonalities Language. It thus can be described as a declarative and bidirectional language that, in contrast to other declarative languages used for model synchronization, does not consist of consistency constraints but of the explicit definition of what information is shared between metamodels. This definition can then be used to derive transformations either between the (then explicitly instantiated) Commonalities metamodel and the metamodels that are to be kept consistent with each other or between those metamodels directly. The Commonalities language implements the first option and generates transformations defined in the Reactions Language.

The *Highly Scalable Incremental Pattern Matching Engine* (*HiPE*) [17] is a pattern-matching engine used by the eMoflon::IBeX TGG framework [38], which is described in section 2.5. HiPE is based on the *Rete Match Algorithm* proposed by Forgy [12], which it implements in a "massively parallelized variant" [30]. By being able to be used as a pattern matcher to find

TGG rule matches in model graphs, HiPE is an alternative to the concept presented in this work. However, it differs from the concept since it is based on detecting patterns on the subgraph that is created when elements are considered that are not yet covered by TGG rule applications, while the pattern-matching concept presented in this work detects patterns in sequences of changes to a model and makes a choice between overlapping matches. Both features are intended to introduce a preference of matches with regard to better reflecting what the user actions behind the changes were.

The Choice between overlapping matches is partially made by using and modifying heuristics of Khelladi et al. [25]. Those are described in section 2.6 and section 3.5.

7. Conclusion

In this work, a concept for using *Triple Graph Grammars* (TGGs) for consistency preservation in *Virtual Single Underlying Models* (V-SUMs) was developed. To evaluate this concept, a prototype was implemented using the Vitruvius framework and *eMoflon::IBeX* and evaluated with regard to correctness, consistency completeness and performance.

The developed concept presented in chapter 3 includes using sequences of *atomic* changes to one model to find rule applications that match the change but also aims to stay close to the user's intention, assuming that the change sequence represents that. To that end, the concept of *Backward-Conversion Pattern Matching* was developed, which revolves around the idea of transforming TGG rules to *Change Sequence Templates*, a data structure that is more easily matchable to single atomic changes in a sequence but also retains the relevant information on the graphical structure of the TGG rule. In chapter 4, details on the implementation of the prototype that realizes this concept are shown, as well as limitations of eMoflon::IBeX. chapter 5 describes the evaluation process, presents and discusses the results. It showed that correctness is no major drawback of the concept, but of the prototype, or, more precisely formulated, of eMoflon::IBeX. Consistency completeness was measured by comparison with the *Reactions* language, and it showed, that in realistic application scenarios, while certain shortcomings could be identified, with 93.33% full and 4.44% partial representability, consistency completeness is also high. It can be assumed that some of these shortcomings can be eliminated by extending the approach to include user interactions, like it is done in the *Reactions* language. Regarding performance, it could be shown that with shorter change sequences, in the evaluation case up to 128 atomic changes, the prototype can keep up with the high-performance pattern matching engine HiPE which is used and recommended by the used TGG framework. From there on upwards, in comparison, it has shown that scalability is not a property of the prototype in its current form, since runtime complexity strongly diverges from HiPE. However, in section 7.2 suggestions to mitigate this weakness are made.

In the following, threats to the validity of the results presented in section 5.3 are presented. Finally, an overview of possible improvements of both the prototype and the concept is given.

7.1. Threats To Validity

Construct Validity As described in the evaluation chapter (see section 5.1), the evaluation was structured using a GQM plan to ensure high construct validity.

External Validity To evaluate how far the approach presented in chapter 3 reaches the goals defined in section 5.1, a prototype was implemented. Considering the inadequacies of the eMoflon::IBeX framework described in subsection 4.5.2, the raw data generated in the evaluation can be seen as a conservative estimation of the approach's capabilities. This has been accounted for by providing an artificial category with an estimation of the results without the mentioned proxy problem. There, each failing change sequence was manually inspected, and it was determined whether they were similar to other change sequences that succeeded. Further, the evaluation has been conducted with the *Java2UML* evaluation case that can be seen as realistic. However, more evaluation cases would further improve external validity, as well as more tests on the same evaluation case. The fact that the *Reactions* language, which is the baseline for consistency completeness in **M2.1.1**, is a Turing-complete imperative language, while the prototype's imperativity is limited to attribute conditions, is not reflected in the evaluation results and can be seen as a threat to external validity. It also can be interpreted as the absence of imperativity/ turing-completeness not being a major drawback for a CPR in realistic scenarios. This ambiguity in interpretation raises the need for further investigation, using more evaluation cases.

Reproducibility *Reproducibility* of the evaluation results is given insofar, as the evaluation process is implemented as *JUnit* tests, which are available as part of the replication package. Reproducing the exact same test results should not be threatened for goals **G1** and **G2**, but since evaluating performance (**G3**) included time measurements with pseudo-random input data, reproducibility is limited to the runtime trend and the comparison to HiPE. The threat of the runtime trend being unstable has been mitigated by running each single test 20 times, and taking the median of all measurements.

Replicability The translation of reactions to TGG rules poses a potential threat to replicability, since that is a subjective process. To mitigate that threat, the evaluation has been gone through systematically, and the rule mappings have been documented. Another threat to replicability is introduced by the artificial category with an estimation of the results without the mentioned proxy problem in metric **M1.1.1**, since deciding what would be correctly matched if the proxy problem did not exist is also a subjective assessment.

7.2. Future work

While developing the concept, implementing the prototype, and evaluating the prototype, various ideas of improvement and extension came up, of which the change propagation could benefit.

Although, mainly in the process of evaluation, some performance optimizations were implemented, there is still room left for further optimization, since that was not the main driving goal of this work. Some of these performance optimizations fall under the category of static precalculations. In the prototype, the conversion process currently is performed each time a change propagation is triggered, which is unnecessary since the resulting change sequence templates only depend on the TGG rules, not on any runtime information. Further, falling partly in the category of static precalculations, patterns that exhibit a containment relation could be found out in a precalculation step and grouped together by extending the change sequence template structure. This might improve both green matching and pattern selection performance. In the former, a change sequence template that structurally consists of the "biggest" pattern, but defines sets of EChange wrappers that represent contained patterns, is thinkable. This would reduce the number of change sequence templates that have to be matched against a change sequence and thus the runtime of green pattern matching.

Since subsection 5.3.3 showed that in comparison to HiPE [17] the approach performs worse from 128 model elements/EChanges upwards (or 512, if coverage flattening is omitted), an approach to split the change sequence in sub-sequences of, e.g., 128 EChanges would be conceivable to improve performance but, of course, limit the complex change detection, especially in the border areas where the change sequence was split. This could again be mitigated by choosing different borders, running the algorithm twice, and choosing the result with more rule applications. While seeming unperformant, this only would introduce a linear overhead, which is negligible compared to the non-linear growth in runtime that the prototype shows. Figure A.1 depicts a projection of this approach which indicates that splitting could drastically improve performance, to the point where it would outperform HiPE if the split size was chosen small enough.

Since the bigger picture is to generate a sequence of changes to the target model with the approach presented in this thesis, that will also have to be implemented. A basic approach would be to calculate that change sequence from state difference, but utilizing the applied patterns might improve reflecting the user intentions in the change sequence. Imagining a constellation where a change is propagated from model A to B to C and both propagation transformations use TGGs, it might even be possible to reuse matches from the A-to-B transformation in the B-to-C- transformation, if there are overlaps between target rules of A-to-B and source rules of B-to-C, which seems not far-fetched in the context of a V-SUMM.

As described in subsection 4.5.2 and subsection 5.3.1, the eMoflon::IBeX framework has bugs and limitations. One reduces the correctness of the consistency preservation process by a great margin, from potentially 95.56% to 57.78% in the evaluation, and another breaks consistency because of imprecise serialization. Thus, using another framework, like eMoflon::Neo [37], might be worth considering moving to.

7.3. Acknowledgements

I thank my first advisor, Lars König, for his support and advice in countless meetings, many emails, and correction iterations of this and other documents. Also, I thank my second advisor, Thomas Weber, for his support in the code and architecture reviews and a long debugging session.

Bibliography

[1] Colin Atkinson, Dietmar Stoll, and Philipp Bostan. “Orthographic Software Modeling: A Practical Approach to View-Based Development”. en. In: *Evaluation of Novel Approaches to Software Engineering*. Ed. by Leszek A. Maciaszek, César González-Pérez, and Stefan Jablonski. Berlin, Heidelberg: Springer, 2010, pp. 206–219. ISBN: 978-3-642-14819-4. DOI: 10.1007/978-3-642-14819-4_15.

[2] Victor R. Basili and David M. Weiss. “A Methodology for Collecting Valid Software Engineering Data”. In: *IEEE Transactions on Software Engineering* SE-10.6 (Nov. 1984). Conference Name: IEEE Transactions on Software Engineering, pp. 728–738. ISSN: 1939-3520. DOI: 10.1109/TSE.1984.5010301.

[3] Guy Caplat and Jean Louis Sourrouille. “Model mapping in MDA”. In: *Workshop in Software Model Engineering (WISME2002).* Vol. 196. Citeseer, 2002.

[4] Fei Chen. *Änderungsgetriebene Konsistenzhaltung zwischen UML-Klassenmodellen und Java-Code. Bachelors's thesis, Karlsruhe Institute of Technology (KIT),* 2017.

[5] K. Czarnecki and S. Helsen. “Feature-based survey of model transformation approaches”. en. In: *IBM Systems Journal* 45.3 (2006), pp. 621–645. ISSN: 0018-8670. DOI: 10.1147/sj.453.0621.

[6] Hartmut Ehrig, Annegret Habel, and Hans-Jörg Kreowski. “Introduction to graph grammars with applications to semantic networks”. en. In: *Computers & Mathematics with Applications* 23.6-9 (Mar. 1992), pp. 557–572. ISSN: 08981221. DOI: 10.1016/0898-1221(92)90124-Z.

[7] *EMF Compare | Home.* URL: https://eclipse.dev/emf/compare/ (visited on 05/09/2025).

[8] *eMoflon (TGG tool) Tutorial.* URL: https://github.com/eMoflon/emoflon-ibex-tutorial/releases/download/v2023.06.26/emoflon-tutorial.pdf.

[9] *eMoflon::IBeX | eMoflon Project Site.* URL: https://emoflon.org/ibex/#feature-overview-2 (visited on 04/18/2025).

[10] *eMoflon/emoflon-ibex-tutorial: Tutorial projects and documentation for eMoflon::IBeX.* URL: https://github.com/eMoflon/emoflon-ibex-tutorial/tree/master (visited on 05/09/2025).

[11] *EObject (EMF Documentation).* URL: https://download.eclipse.org/modeling/emf/emf/javadoc/2.9.0/org/eclipse/emf/ecore/EObject.html (visited on 05/10/2025).

[12] Charles L. Forgy. “Rete: A fast algorithm for the many pattern/many object pattern match problem”. In: *Artificial Intelligence* 19.1 (Sept. 1982), pp. 17–37. ISSN: 0004-3702. DOI: 10.1016/0004-3702(82)90020-0.

[13] Lars Fritsche et al. “Advanced Consistency Restoration with Higher-Order Short-Cut Rules”. en. In: *Graph Transformation.* Ed. by Maribel Fernández and Christopher M. Poskitt. Cham: Springer Nature Switzerland, 2023, pp. 184–203. ISBN: 978-3-031-36709-0. DOI: `10.1007/978-3-031-36709-0_10`.

[14] Lars Fritsche et al. “Avoiding unnecessary information loss: correct and efficient model synchronization based on triple graph grammars”. en. In: *International Journal on Software Tools for Technology Transfer* 23.3 (June 2021), pp. 335–368. ISSN: 1433-2787. DOI: `10.1007/s10009-020-00588-7`.

[15] Lars Fritsche et al. “Short-Cut Rules”. en. In: *Software Technologies: Applications and Foundations.* Ed. by Manuel Mazzara, Iulian Ober, and Gwen Salaün. Cham: Springer International Publishing, 2018, pp. 415–430. ISBN: 978-3-030-04771-9. DOI: `10.1007/978-3-030-04771-9_30`.

[16] David Harel and Bernhard Rumpe. *Modeling languages: Syntax, semantics and all that stuff.* Tech. rep. Citeseer, 2000. URL: `https://citeseerx.ist.psu.edu/document?repid=rep1&type=pdf&doi=072663058126978f4b6478121c71de30e404160c` (visited on 09/24/2024).

[17] HiPE-DevOps. *Highly (Scalable) Incremental Pattern matching Engine (HiPE).* original-date: 2019-07-31T13:27:23Z. May 2022. URL: `https://github.com/HiPE-DevOps/HiPE-Updatesite` (visited on 05/01/2025).

[18] *https://www.omg.org/spec/UML/2.5.1/PDF.* URL: `https://www.omg.org/spec/UML/2.5.1/PDF` (visited on 09/24/2024).

[19] John Hutchinson, Mark Rouncefield, and Jon Whittle. “Model-driven engineering practices in industry”. In: *Proceedings of the 33rd International Conference on Software Engineering.* ICSE ’11. New York, NY, USA: Association for Computing Machinery, May 2011, pp. 633–642. ISBN: 978-1-4503-0445-0. DOI: `10.1145/1985793.1985882`.

[20] John Hutchinson, Jon Whittle, and Mark Rouncefield. “Model-driven engineering practices in industry: Social, organizational and managerial factors that lead to success or failure”. In: *Science of Computer Programming.* Special issue on Success Stories in Model Driven Engineering 89 (Sept. 2014), pp. 144–161. ISSN: 0167-6423. DOI: `10.1016/j.scico.2013.03.017`.

[21] *International Organization for Standardization, 2011. Systems and Software Engineering – Architecture Description (ISO/IEC/IEEE 42010:2011(E)).* en.

[22] *JDK 21 Documentation - Home.* en. URL: `https://docs.oracle.com/en/java/javase/21/` (visited on 05/10/2025).

[23] *JUnit 5.* URL: `https://junit.org/junit5/` (visited on 05/10/2025).

[24] Stuart Kent. “Model Driven Engineering”. en. In: *Integrated Formal Methods.* Ed. by Michael Butler, Luigia Petre, and Kaisa Sere. Berlin, Heidelberg: Springer, 2002, pp. 286–298. ISBN: 978-3-540-47884-3. DOI: `10.1007/3-540-47884-1_16`.

[25] Djamel Eddine Khelladi et al. “Detecting Complex Changes During Metamodel Evolution”. en. In: *Advanced Information Systems Engineering*. Ed. by Jelena Zdravkovic, Marite Kirikova, and Paul Johannesson. Cham: Springer International Publishing, 2015, pp. 263–278. ISBN: 978-3-319-19069-3. DOI: 10.1007/978-3-319-19069-3_17.

[26] Ekkart Kindler and Robert Wagner. *Triple Graph Grammars: Concepts, Extensions, Implementations, and Application Scenarios*. en. Tech. rep.

[27] Heiko Klare and Joshua Gleitze. “Commonalities for Preserving Consistency of Multiple Models”. In: *2019 ACM/IEEE 22nd International Conference on Model Driven Engineering Languages and Systems Companion (MODELS-C)*. Sept. 2019, pp. 371–378. DOI: 10.1109/MODELS-C.2019.00058.

[28] Heiko Klare et al. “Enabling consistency in view-based system development — The Vitruvius approach”. In: *Journal of Systems and Software* 171 (2021), p. 110815. ISSN: 0164-1212. DOI: https://doi.org/10.1016/j.jss.2020.110815.

[29] Max Emanuel Kramer. *Specification Languages for Preserving Consistency between Models of Different Languages*. de. 2017. DOI: 10.5445/IR/1000069284.

[30] Maximilian Kratz et al. “Model-Driven Rapid Prototyping for Control Algorithms with the GIPS Framework (System Description)”. en. In: *Electronic Proceedings in Theoretical Computer Science* 417 (Mar. 2025), pp. 157–172. ISSN: 2075-2180. DOI: 10.4204/EPTCS.417.9.

[31] *OMG Document – ormsc/14-06-01 (MDA Guide revision 2.0)*. URL: http://www.omg.org/cgi-bin/doc?ormsc/14-06-01 (visited on 09/23/2024).

[32] Andy Schürr. “Specification of graph translators with triple graph grammars”. en. In: *Graph-Theoretic Concepts in Computer Science*. Ed. by Ernst W. Mayr, Gunther Schmidt, and Gottfried Tinhofer. Berlin, Heidelberg: Springer, 1995, pp. 151–163. ISBN: 978-3-540-49183-5. DOI: 10.1007/3-540-59071-4_45.

[33] Herbert Stachowiak. *Allgemeine Modelltheorie*. de. Springer, 1973. ISBN: 978-3-211-81106-1.

[34] Dave Steinberg et al. *EMF: eclipse modeling framework*. Pearson Education, 2008. URL: https://books.google.de/books?hl=en&lr=&id=sA0zOZuDXhgC&oi=fnd&pg=PT20&dq=EMF:+Eclipse+Modeling+Framework+Steinberg&ots=2LJKXVXkIp&sig=uUG9wKW8bSbFFP47IpWYJ1mUwmQ (visited on 09/27/2024).

[35] *Vitruv-Change/bundles/tools.vitruv.change.correspondence/metamodel/correspondence.ecore at main · vitruv-tools/Vitruv-Change*. en. URL: https://github.com/vitruv-tools/Vitruv-Change/blob/main/bundles/tools.vitruv.change.correspondence/metamodel/correspondence.ecore (visited on 10/09/2024).

[36] *vitruv-tools/Vitruv-CaseStudies: Case Studies for the Vitruvius Framework*. URL: https://github.com/vitruv-tools/Vitruv-CaseStudies (visited on 05/10/2025).

[37] Nils Weidmann and Anthony Anjorin. “eMoflon::Neo - Consistency and Model Management with Graph Databases”. en. In: *STAF workshops* (2021), pp. 54–64.

[38] Nils Weidmann et al. “Incremental Bidirectional Model Transformation with eMoflon::IBeX”. en. In: *Graph Transformation: 12th International Conference, ICGT 2019, Held as Part of STAF 2019, Eindhoven, The Netherlands, July 15–16, 2019, Proceedings 12* (2019), pp. 131–140.

[39] Jon Whittle, John Hutchinson, and Mark Rouncefield. “The State of Practice in Model-Driven Engineering”. In: *IEEE Software* 31.3 (May 2014), pp. 79–85. ISSN: 1937-4194. DOI: `10.1109/MS.2013.65`.

A. Appendix

A.1. First Appendix Section

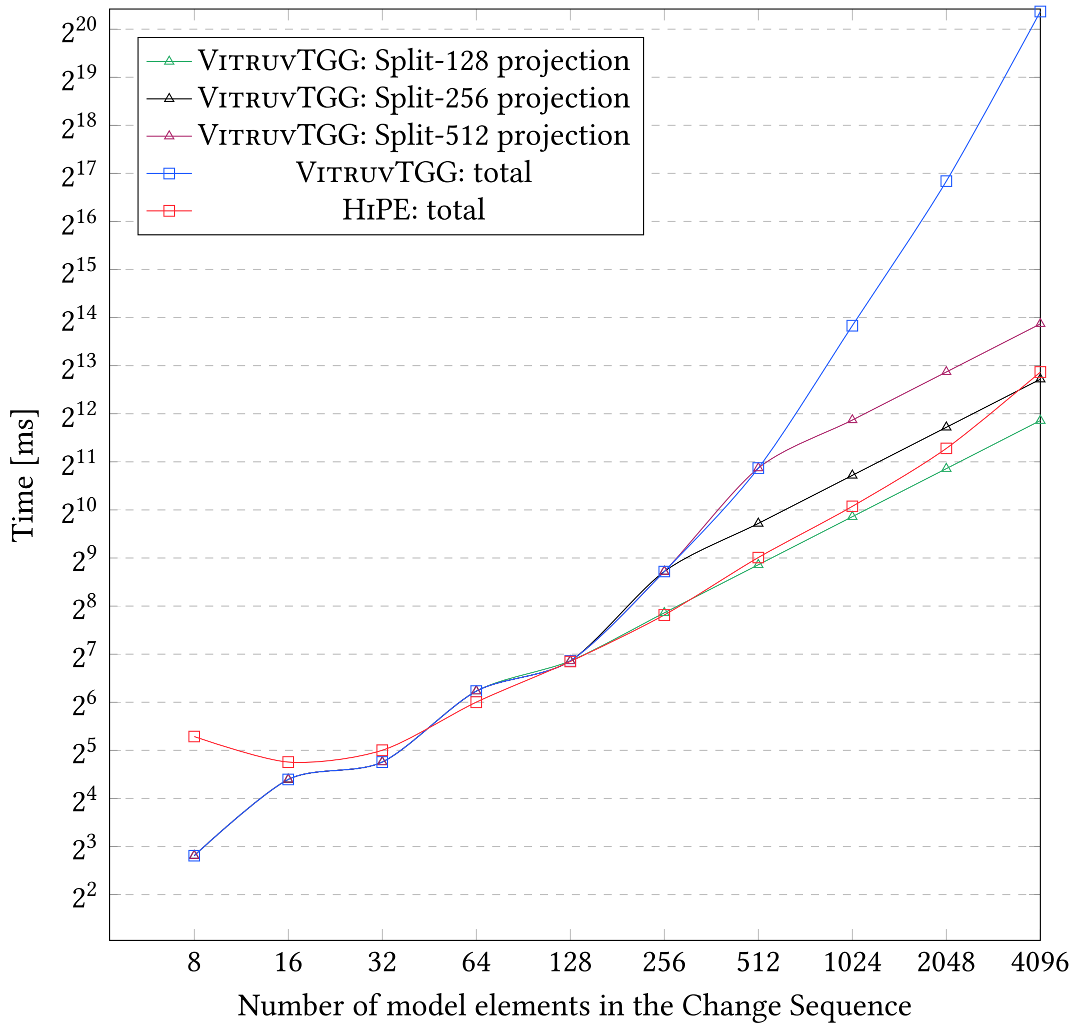

Figure A.1.: Runtime complexity projection trend of VITRUVTGG with split change sequence and HIPE, different split sizes.