A Systematic Approach and Guidelines to Developing a Triple Graph Grammar

Anthony Anjorin
Chalmers | University of Gothenburg
anjorin@chalmers.se

Erhan Leblebici, Roland Kluge, Andy Schürr
Technische Universität Darmstadt
{firstname.lastname}@es.tu-darmstadt.de

Perdita Stevens
University of Edinburgh
perdita.stevens@ed.ac.uk

Abstract

Engineering processes are often inherently concurrent, involving multiple stakeholders working in parallel, each with their own tools and artefacts. Ensuring and restoring the consistency of such artefacts is a crucial task, which can be appropriately addressed with a bidirectional transformation (*bx*) language. Although there exist numerous *bx* languages, often with corresponding tool support, it is still a substantial challenge to learn how to actually *use* such *bx* languages. Triple Graph Grammars (TGGs) are a fairly established *bx* language for which multiple and actively developed tools exist. Learning how to master TGGs is, however, currently a frustrating undertaking: a typical paper on TGGs dutifully explains the basic "rules of the game" in a few lines, then goes on to present the latest groundbreaking and advanced results. There do exist tutorials and handbooks for TGG tools but these are mainly focussed on how to use a particular tool (screenshots, tool workflow), often presenting exemplary TGGs but certainly not how to derive them systematically. Based on 20 years of experience working with and, more importantly, explaining *how* to work with TGGs, we present in this paper a systematic approach and guidelines to developing a TGG from a clear, but unformalised understanding of a *bx*.

1 Introduction and Motivation

Formalizing and maintaining the consistency of multiple artefacts is a fundamental task that is relevant in numerous domains [6]. Bidirectional transformation (*bx*) languages address this challenge by supporting bidirectional change propagation with a clear and precise semantics, based on a central notion of consistency specified with the *bx* language. Although numerous *bx* approaches exist [19], often with corresponding tool support, mastering a *bx* language is still a daunting task. Even once the user has a clear, though unformalised, understanding of what the *bx* should do, embodying this understanding in a *bx* is non-trivial. Formal papers do not address this issue, and neither, usually, do tool-centric handbooks.

In: A. Cunha, E. Kindler (eds.): Proceedings of the Fourth International Workshop on Bidirectional Transformations (Bx 2015), L'Aquila, Italy, July 24, 2015, published at http://ceur-ws.org

Triple Graph Grammars (TGGs) [18] are a prominent example of a *bx* language for which this observation holds: a typical paper on TGGs dutifully explains how TGGs "work" in a few lines, then conjures up a complete and perfect TGG specification for the running example out of thin air. This is not how things work in practice; going from a clear, but unformalised understanding of consistency to a formal TGG specification is difficult, especially for developers who do not already have ample practice with rule-based, declarative, (graph) pattern-based languages. Existing tutorials, handbooks, and introductory papers on TGGs do not alleviate this situation as they are either tool-specific, focusing on how to use a certain TGG tool, or present basic concepts without providing a systematic approach to how TGG rules can be iteratively engineered.

To the best of our knowledge, the only existing work in this direction is from Kindler and Wagner [13] and later, treated in some more detail in Wagner's PhD thesis [22]. Similarly to [21] for model transformation, Kindler and Wagner propose an algorithm that "synthesizes" TGG rules from a set of consistent examples provided by the user. Although this can be very helpful in scenarios where rather simple but numerous TGG rules are required, we have observed that a typical TGG involves a careful design process with direct consequences for the resulting behaviour of derived model synchronizers. Indeed, Kindler and Wagner mention that synthesized TGGs probably have to be adjusted, extended and finalized manually, but do not provide adequate guidance of how this can be done systematically.

Based on 20 years' experience working together with industrial partners and students, learning how to understand and use TGG specifications, our contribution in this paper is, therefore, to provide a systematic approach to creating and extending TGGs. Our aim with the resulting step-by-step process, is to substantially lower the initial hurdle of concretely working with TGGs, especially for other users and researchers in the *bx* community.

The rest of the paper is organized as follows: Sect. 2 reviews the preliminaries of TGGs and provides our running example that is available as `Ecore2Html` from the *bx* example repository [5], and as a plug-and-play virtual machine hosted on *Share*.[1] Sect. 3 introduces an iterative approach and a set of guidelines for engineering a TGG, constituting our main contribution in this paper. Sect. 4 complements this by providing an intuitive understanding of TGG-based synchronization algorithms. Sect. 5 compares our work to other related contributions. Sect. 6 summarizes and gives a brief outlook on future tasks.

2 Running Example and Preliminaries

The running example used in the rest of this paper is a bidirectional transformation between class diagrams and a corresponding HTML documentation. Consider, for instance, a software development project for implementing a TGG-based tool. The project consists of two groups of developers: (i) core developers who are responsible for establishing and maintaining the main data structures and API provided by the tool, and (ii) other developers who mainly use the API and work with the tool (functioning as beta testers for the project). The latter group does not *define* the data structures involved, but is probably in a better position to *document* all packages, classes, and methods. It thus makes sense to maintain two types of artefacts for the two groups of stakeholders: (1) class diagrams, maintained by core developers, and (2) HTML documents, maintained by API users in, e.g., a wiki-like manner. The class diagram to the left of Fig. 1 depicts the main data structures of a TGG tool, with a corresponding folder structure containing HTML files to the right.

The outermost package `TGGLanguage` corresponds to the top-most folder `TGGLanguage`, and contains subpackages and classes, which correspond to subfolders and HTML files, respectively. `TripleGraphGrammar`s consist of `TGGRule`s, which are in essence graph patterns (referred to as `StoryPattern`s) consisting of variables that stand for objects (`ObjectVariable`s) and links (`LinkVariable`s) in a model. This general concept of a graph pattern is extended for TGGs by attaching a `Domain` to each concept. Each `Domain` references a `Metamodel`, a wrapper for the supported metamodelling standard.[2] The classes `StoryPattern`, `LinkVariable`, `ObjectVariable`, and `EPackage` are imported from other class diagrams (greyed out) and are not part of the documentation of `TGGLanguage`. The corresponding documentation model is much simpler: it consists of a folder structure mirroring the package structure in the class diagram, and an HTML file for each class. Note that packages are also documented in corresponding files, which start with "__" to distinguish them from class documentation files (e.g., `__TGGLanguage.html`). Inheritance and references are irrelevant for the documentation model, but all methods of each class (excluding inherited methods) are to be documented as a table with two columns (name of the method and documentation) in the corresponding class documentation file.

[1] `http://is.ieis.tue.nl/staff/pvgorp/share/?page=LookupImage&bNameSearch=eMoflon`

[2] In this case `EPackage` for *Ecore*, the *de facto* standard of the Eclipse Modeling Framework (EMF).

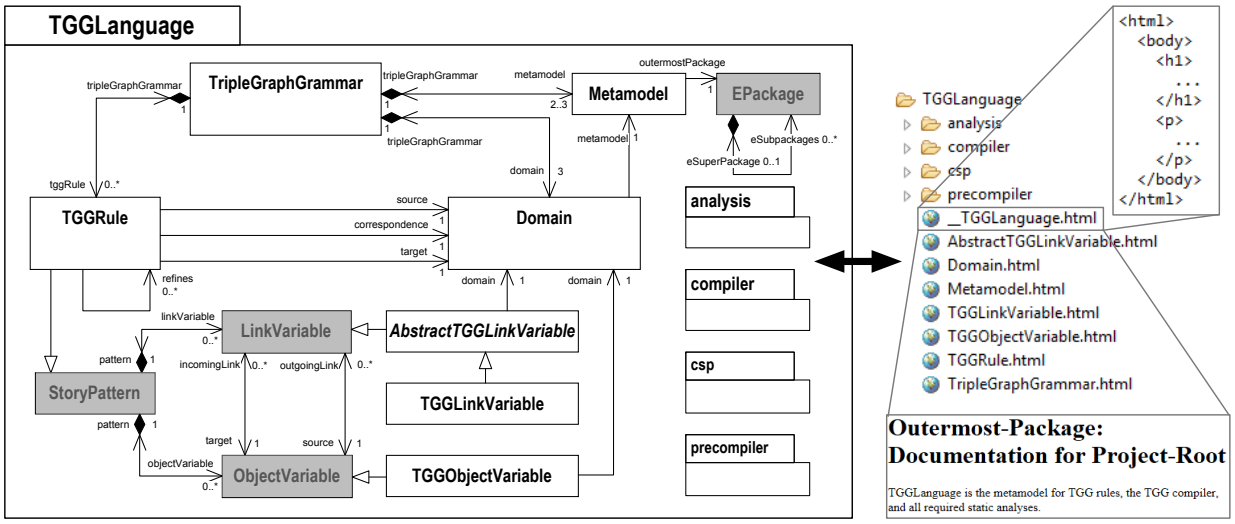

Figure 1: A class diagram and its corresponding HTML documentation

The following points are noteworthy: (1) Information loss is incurred when transforming class diagrams to documentation models (e.g., inheritance relations and references) *and* when transforming documentation models to class diagrams (all entered documentation is lost!). This is the main reason why the change propagation strategy required for this example *must* take the old version of the respective output artefact into account. (2) Not all possible changes make sense: it is arguable whether API users should be able to rename elements in the documentation and propagate such changes to the corresponding class diagrams. Even more arguable are changes such as adding new elements in the documentation that do not yet exist in the class diagram. Such changes may be interpreted as feature requests, but are probably not primary use cases.

Models, Metamodels, Deltas, and Triple Graph Grammars: We assume a very basic understanding of TGGs and of Model-Driven Engineering (MDE) concepts, and refer readers having a hard time understanding the following to, e.g., the eMoflon handbook[3] for a gentle, tutorial-like introduction. Readers more interested in a formal and detailed introduction to TGGs are referred to, e.g., [1].

In the following, we introduce core terms and notation as they are to be understood (informally) in this paper. Based on this understanding, we then propose a systematic, iterative TGG *development process* in Sect. 3, together with a set of *guidelines* or best practices. This process is then demonstrated by applying it to develop a TGG for our running example.

A *model* is an abstraction (a simplification) of something else, chosen to be suitable for a certain task. In an MDE context, *metamodels*, essentially simplified UML class diagrams, are used to define languages of models.

Given two languages of models and their respective metamodels, say a *source* and a *target* language, a *consistency relation* over source and target models is a set of pairs of a source and a target model, which are to be seen as being consistent with each other.

Given a consistent pair of source and target models, a change applied to the source model is referred to as a *source delta*, and a change to the target model is called a *target delta*. As models are graph-like structures consisting of attributed nodes and edges, we may decide that *atomic deltas* are one of: element (node/edge) creation, element deletion, and attribute changes. Atomic deltas can be composed to yield a *composite delta*. A composite delta that only involves element creation is called a *creating delta*. A source/target delta that does nothing is called an *idle source/target delta*, respectively.

Given a consistent pair of source and target models, and a source delta, the task of computing a corresponding target delta that restores consistency by changing the target model appropriately is referred to as *forward model synchronization*, or just model synchronization. This applies analogously to target deltas and computed source deltas, i.e., *backward model synchronization*. The more general task of restoring consistency given both a source and a target delta is referred to as *model integration*, which is outside the scope of this paper.

A *Triple Graph Grammar* (TGG) is a finite set of *rules* that each pair a creating source/target delta with either a corresponding creating target/source delta or with an idle target/source delta, respectively.

[3]Available from `www.emoflon.org`

Each TGG rule states that applying the specified pair of deltas together will extend a given pair of source and target models consistently. To keep track of correspondences between consistent source and target models on an element-to-element basis, a third *correspondence model* can be maintained consisting of explicit *correspondence elements* connecting certain source and target elements with each other. These correspondence elements are typed with a correspondence metamodel, which can be chosen as required. In general, a TGG can thus be used to generate a language of *triples*, consisting of connected source, correspondence, and target models. A triple is denoted as $G_S \leftarrow G_C \rightarrow G_T$ (G for *graph*). A TGG induces a consistency relation as follows: A pair of source and target models (G_S, G_T) is *consistent* if there exists a triple $G_S \leftarrow G_C \rightarrow G_T$ that can be generated by applying a sequence of deltas, as specified by the rules of the TGG.

A *TGG tool* is able to perform model synchronization using a given TGG as the specification of consistency. A TGG tool is *correct* if it only produces results that are consistent according to the given TGG used to govern the synchronization process [18]. As correctness does not in any way imply that information loss should be avoided, one TGG tool is said to be more *incremental* than another if it handles (avoids) information loss better [1]. A TGG tool is said to *scale* if the runtime for synchronization depends polynomially (and not exponentially) on model and delta size, and is *efficient* if the runtime for synchronization only depends on delta size (and no longer on model size) [1].

Example: To explain these basics further, Fig. 2 depicts the source and target metamodels chosen for the running example. The source metamodel (to the left) is *Ecore*, a well-known standard for simple class diagrams. Class diagrams basically consist of packages (`EPackage`s) containing subpackages, and classes (`EClass`es) with methods (`EOperation`s). The target metamodel (to the right) is *MocaTree*, a generic tree-like target metamodel providing concepts to represent a hierarchy of folders with files, where each file contains a tree consisting of `Node`s. Objects of type `Text` but not of type `Node` are used to represent leaves in the tree that cannot have children. In this manner, it is possible to represent arbitrary (X)HTML trees. The correspondence metamodel consists of types connecting source and target elements as required for the rules and is omitted here.

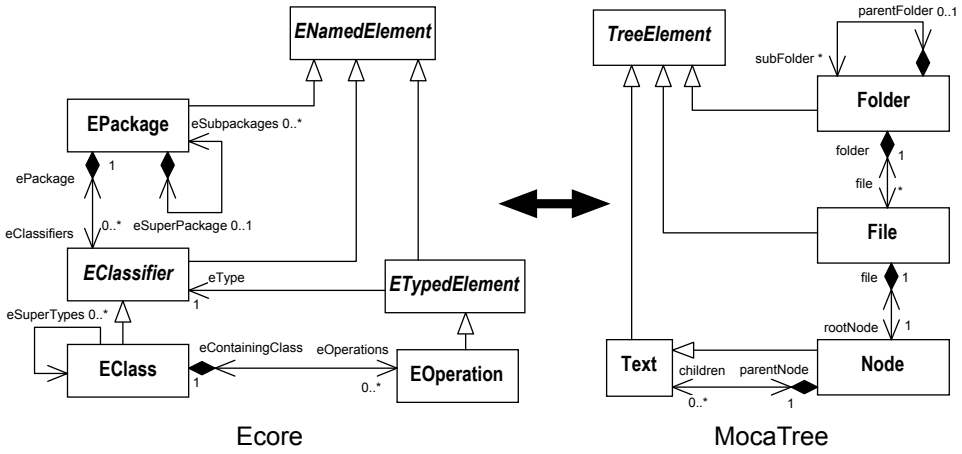

Figure 2: Source and target metamodels for the running example

To introduce the notation and language features used in this paper to specify TGG rules, let us consider two TGG rules for handling (sub)packages and their corresponding documentation. The TGG rule for handling *root* packages is depicted to the left of Fig. 3. The source and target deltas paired by this rule are: (i) creating a new `EPackage` in the source model, and (ii) creating a `Folder` with a single HTML `File`. The basic structure of the HTML file, consisting of a header and a paragraph where a description of the package can be entered, is also created in the same target delta. The package and folder are connected with a correspondence node of type `EPackageToFolder`. In standard TGG visual notation, all elements *created* in a rule are depicted in green with an additional `"++"` markup for black and white printouts. Correspondence nodes are additionally depicted as hexagons to clearly differentiate them from source and target elements.

A set of *attribute constraints* specifies the relationship between attribute values of the elements in a TGG rule. All TGG tools provide an extra, typically simple textual language for expressing such attribute constraints. In this case, the `eq` constraint expresses that `ePackage` has the same name as `docuFolder`. To express that the name of the `htmlFile` should start with `"__"`, end with `".html"`, and contain the name of `ePackage`, two attribute constraints `addSuffix` and `addPrefix` are used, where `withSuffix` is a temporary variable.

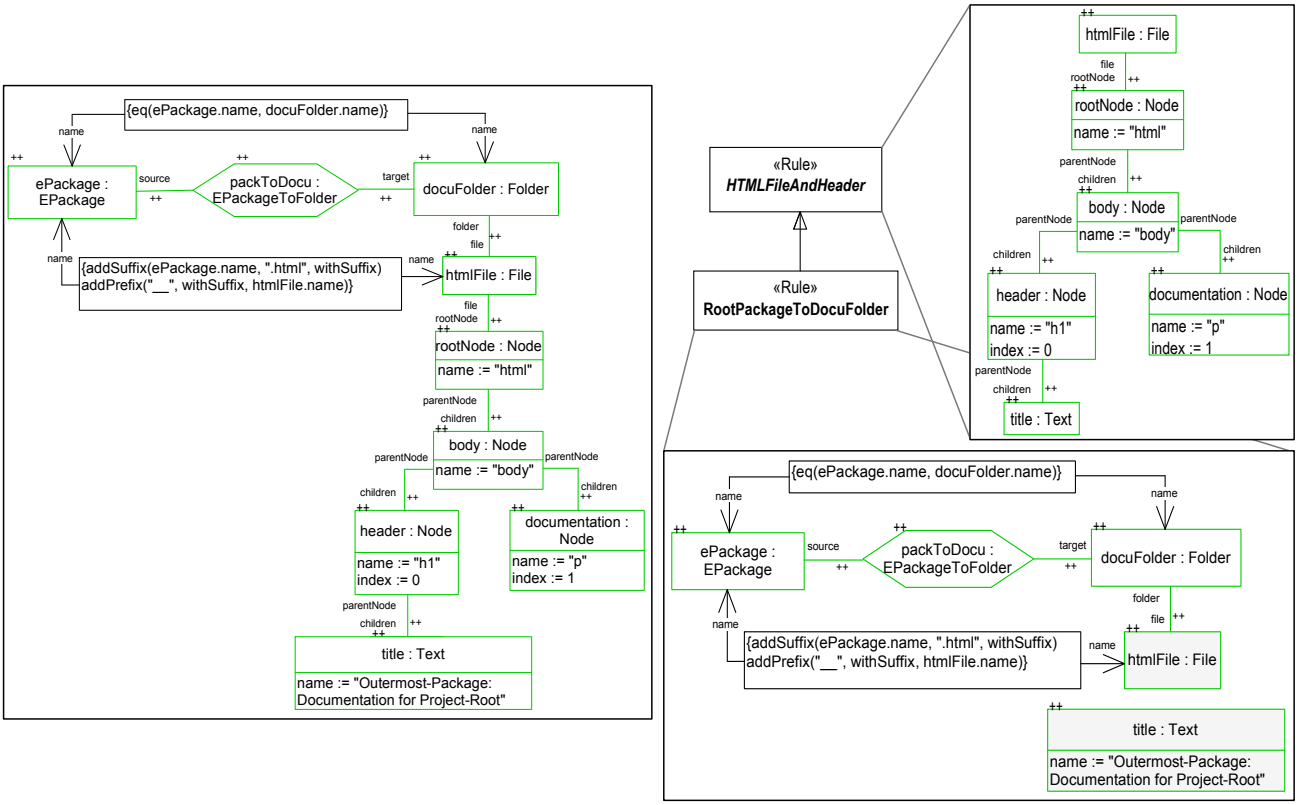

Figure 3: Handling root packages: without (left) and with (right) rule refinement

For cases where an attribute is simply assigned a constant, this can be "inlined" in the respective node, e.g., `name := "html"` in `rootNode`. If we consider Fig. 1 again, we can now compare the creating deltas in the TGG rule to the package `TGGLanguage`, the folder `TGGLanguage` and the HTML file `__TGGLanguage.html`, and see that all attribute constraints are fulfilled.

It is often useful to extract parts of a rule into a *basis rule*, so that these parts can be reused in other *subrules*. Apart from possibly enabling reuse, readability can also be increased by decomposing a rule into modular fragments that each handle a well-defined concern. Many TGG tools support some form of *rule refinement* [3, 8, 14] as depicted to the right of Fig. 3: A new TGG rule `HTMLFileAndHeader`, creating the basic structure of an HTML file, has been extracted and is now *refined* (denoted by an arrow) by the subrule `RootPackageToDocuFolder` to yield the same rule as depicted to the left of Fig. 3. As we do not want to allow creating basic HTML files on their own, `HTMLFileAndHeader` is declared to be *abstract* (denoted by displaying the rule's name in italics). This means that it is solely used for modularization and not for synchronization. The exact details of rule refinement are out-of-scope for this paper, but in most cases (as here), it is a merge of the basis with the refining rule, where elements in the refining rule override elements with the same name in the basis rule (this is the case for `htmlFile` and `title`). Such overriding elements are shaded light grey in subrules to improve readability. TGGs with rule refinements are flattened to normal TGGs. Abstract rules are used in the process, but are not included in the final TGG. Non-abstract rules are called *concrete* rules.

Fig. 4 depicts a further TGG rule `SubPackageToDocuFolder` for handling subpackages (`EPackage`s with a super package). `SubPackageToDocuFolder` refines `RootPackageToDocuFolder` and additionally places the created package into a super package, and the corresponding subfolder into a parent folder (determined using the correspondence node `superToFolder`). Finally, the title is adjusted for subpackages by overriding it in `SubPackageToDocuFolder`. This rule shows how *context* can be demanded; the *context elements* `superPackage`, `superToFolder`, `superFolder`, and connecting `source` and `target` links are depicted in black and must be present (created by some other rule) before the rule can be applied. In this sense, the context elements form a *precondition* for the application of the rule.

Our current TGG consists of two concrete rules and one abstract rule, and can be used to generate consistent package hierarchies and corresponding folder structures with HTML files for documenting the packages.

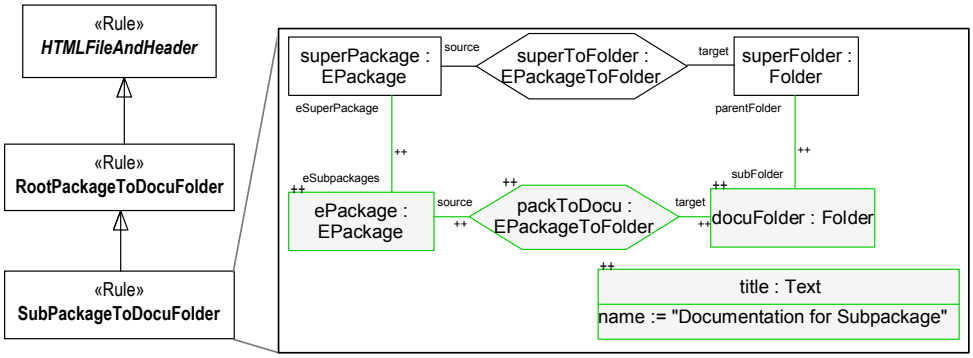

Figure 4: Handling subpackages and their documentation

In the following section, we shall take a look at how to systematically develop TGG rules in a step-by-step process, discussing various *kinds* of TGG rules and how design choices affect derived TGG-based synchronizers.

3 An Iterative Process and Guidelines for Developing a TGG

To introduce basic concepts we have already specified a TGG to handle package structures and their HTML documentation. This was done in a relatively ad-hoc fashion, choosing source and target metamodels, deciding to start with handling packages, and specifying the required TGG rules without consciously applying any systematic process. Although this might work fine for simple cases or for seasoned TGG experts, we propose the following process depicted in Fig. 5, which consists of four steps outlined in the following sections. In each step, we give guidelines that represent current best practice and provide guidance when making design decisions.

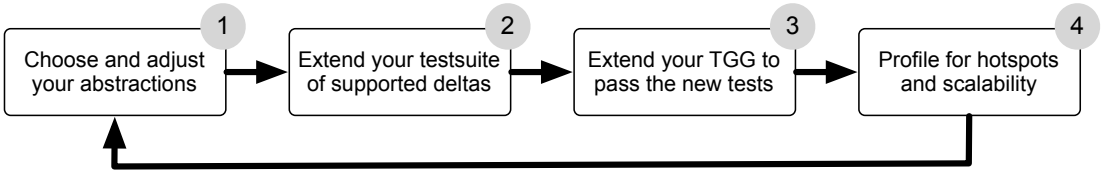

Figure 5: Developing a TGG with an iterative, test-driven development process

3.1 Choose and adjust your abstractions (metamodels)

In practice, the final artefacts to be synchronized are typically fixed for a certain application scenario, e.g., XML files, a certain textual format, a tool's API, or programs in some programming language. As TGG tools do not usually operate directly on these final artefacts, a parser/unparser component, referred to as an *adapter* in the following, is required to produce a graph-like abstraction of the artefacts. The choice of source/target metamodels thus becomes a degree of freedom with a direct impact on the complexity of the required adapter.

Guideline 1 (Adapter complexity vs. rule complexity)**.** *Choosing very high-level source and target metamodels might lead to simple, elegant TGG rules but also requires complex adapters in form of (un)parsers or some other import/export code. As this shifts complexity away from the TGG, strive to keep such pre/post-processing adapter logic to a minimum. Decomposing the synchronization into multiple steps by introducing an intermediate metamodel and developing two TGGs instead of one can be a viable alternative.*

Example: For our running example, the source artefacts are XMI files representing class diagrams, while target artefacts are XHTML files. As the class diagrams already conform to Ecore, choosing Ecore as the source metamodel means that the standard EMF XMI reader and writer can be used. To handle XHTML files, a simple and generic XML adapter is used that produces instances of a basic, tree-like metamodel `MocaTree`, as depicted in Fig. 2. Although these choices reduce the required adapter logic to a minimum (G1), our TGG rules are rather verbose, especially in the target domain (cf. Fig. 3). We addressed this with rule refinement (any form of modularity concept could be beneficial in this case if supported by the chosen TGG tool), but could also have chosen a richer target metamodel and shifted most of the complexity to a problem-specific XHTML parser and unparser instead.

3.2 Extend your test suite of supported deltas

Although it is generally accepted best practice in software development to work iteratively and to apply regression testing, beginners still tend to specify multiple rules or even a complete TGG without testing. This is particularly problematic because the understanding of the consistency relation often grows and changes *while* specifying a TGG. The following guideline highlights that any non-trivial TGG should be supported by a test suite.

Guideline 2 (Take an iterative, test-driven approach)**.** *When specifying a TGG, take a test-driven approach to prevent regression as the rules are adjusted and extended. Run your test suite after every change to your TGG.*

In this context, a source *test case* consists of: (i) a consistent triple (possibly empty), (ii) a source delta to be applied to the triple, and (iii) a new target model representing the expected result of forward synchronizing the source delta. Target test cases are defined analogously. Although all important deltas for an application scenario should eventually be tested, it makes sense to focus first on creating deltas as these can be almost directly translated into TGG rules.

Guideline 3 (Think in terms of creating source and target deltas)**.** *Think primarily in terms of consistent pairs of creating source and target deltas and derive corresponding test cases. This simplifies the transition to a TGG.*

The following two guidelines propose to handle "easy" cases first before going into details. Concerning creating deltas, "easy" translates to "not dependent on context".

Guideline 4 (Start with context-free creating deltas)**.** *Start with context-free pairs of creating deltas as these are usually simpler. A context-free delta can always be propagated in the same way, independent of what the elements it creates will be later connected with.*

Although graphs do not have a "top" or "bottom" in general, in many cases models do have an underlying containment tree. Top-down thus means container before contents. As containers typically do not depend on their contents, we get the following guideline as a corollary of Guideline 4:

Guideline 5 (Prefer top-down to bottom-up)**.** *If possible, start top-down with roots/containers of the source and target metamodels as containers typically do not depend on their contents.*

Example: Applying these guidelines to our running example, we chose to handle package hierarchies in a first iteration (G2). We started thinking about creating a root package in the class diagram as a source delta, whose corresponding target delta is creating a folder containing a single HTML file named "__" + `ePackage.name`, which is to contain the documentation for the root package (G3). This is always the case irrespective of what the root package/folder later contains and is thus a context-free pair of creating deltas (G4). We took a top-down approach (G5) handling root packages before subpackages (cf. Fig. 3 and Fig. 4).

3.3 Extend your TGG to pass the new tests

After discussing with domain experts and collecting test cases addressing a certain aspect, e.g., (sub)packages and folders, the next step is to specify new, or adjust existing, TGG rules to pass these tests. To accomplish this, it is crucial to understand that there are only a few basic *kinds* of TGG rules as depicted schematically in Fig. 6. The names are chosen to give a geographical intuition, where green and black clouds/arrows denote created and context elements, respectively:

Islands are context-free rules that do not demand any context. An island may either be idle, creating either source or target elements, or it may create both, source *and* target elements. After applying such a rule, an isolated "island" of elements is created.

Extensions require a context island and then *extend* it by creating and directly attaching new elements to it.

Bridges connect two context islands by creating new elements to form a *bridge* between the islands. Bridges can of course be generalized to connect multiple islands, but remain fundamentally the same. Bridges connecting more than two islands are rare in practice, probably because they become increasingly complex to comprehend.

Ignore rules (not depicted) are TGG rules that do not create any elements in either the source or target domain. Such rules state explicitly that applying a certain creating delta in one domain has no effect on the other domain.

Based on these basic kinds of TGG rules, we propose the following guidelines.

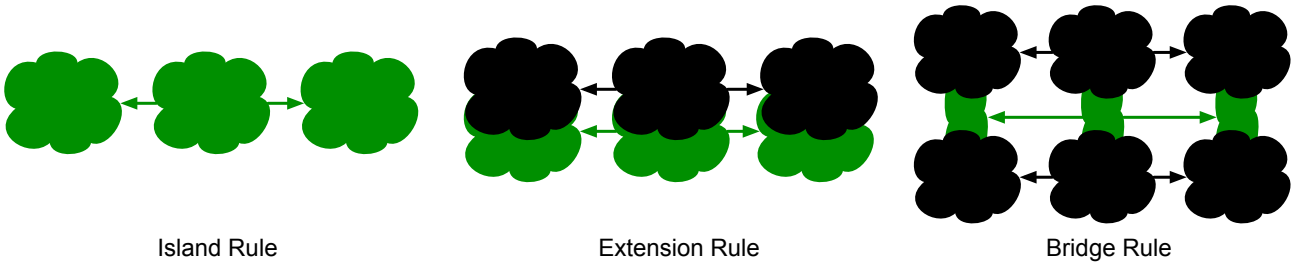

Figure 6: Different kinds of TGG rules: *Islands* (left), *Extensions* (middle), and *Bridges* (right)

Guideline 6 (Prefer islands and bridges to extensions)**.** *When specifying an extension, always ask domain experts if it is acceptable to split the extension into an island and a bridge, as this often improves derived synchronizers. Extensions should only be used if an island would have to change once it is connected via a bridge.*

An extension states that all created elements are dependent on the specified context. In many cases, this introduces unnecessary dependencies between islands. In general, TGG-based synchronization works better, the fewer context dependencies are specified in rules.

Guideline 7 (Formalize information loss via explicit ignore rules)**.** *Most TGG tools attempt to ignore irrelevant elements automatically, as specifying this explicitly with ignore rules can be tedious, especially in the first few iterations, when the TGG covers only a small subset of the source and target metamodels. Explicit ignore rules are nonetheless better than tool-specific heuristics and automatic ignore support, and should be preferred.*

Example: Applying these guidelines to our running example, we now discuss the current TGG (handling package/folder hierarchies) as well as new rules for handling classes and methods, together with their respective documentation. Reflecting on rules `RootPackageToDocuFolder` and `SubPackageToDocuFolder` (cf. Fig 3 and 4) now in the light of our guidelines, we can identify `RootPackageToDocuFolder` as an island, and `SubPackage-ToDocuFolder` as an extension thereof. Can we split the extension into an island and a bridge (G6)? Although subpackages are treated just like root packages, it must be clear from the corresponding documentation file that this is the documentation for a subpackage and not a root package. The heading in the file (`title` in `Root-PackageToDocuFolder`) must, therefore, be adjusted appropriately. Interestingly, if a (sub)package `p` is deleted in the source model, all subpackages of `p` consequently become root packages and must now be (re)translated differently! This also applies to making a former root package a subpackage by connecting it to a super package. In this case, the root package is now a subpackage and must also be (re)translated appropriately. Creating and documenting subpackages in this manner is an example of creating deltas that *cannot* be handled as an island.

Creating a class `c` corresponds to creating an HTML file named `<c.name>` + ".html", this time with a header "EClass" + `<c.name>`. In addition, an empty HTML table to contain all methods of the class should be created in the file. Adding a class `c` to a package `p` corresponds to simply placing the documentation file for `c` into the documentation folder for `p`. This time around, these consistency requirements can be transferred directly to an island and a bridge: Fig. 7 depicts the island `EClassToHTMLFile` for handling the creation of an `EClass` and its documentation file with internal HTML structure. `EClassToHTMLFile` once again refines `HTMLFileAndHeader` to reuse the basic structure of an HTML file. The `body` of the HTML file is extended, however, by a table node `methodTable` and an extra subheader for the table. Attribute constraints are used to ensure that the name and header of the file correspond appropriately to the created class. Specifying this as an island means that classes and their documentation are created in this manner, independent of what package the class is later placed in.

Fig. 7 also depicts the bridge `EClassToFileBridge` used to handle adding classes to packages and their documentation files to the corresponding documentation folder. In this case the "bridge" consists of an edge in each domain. In general, however, an arbitrary number of elements can be created to connect the context clouds as required. The primary advantage of using a bridge here instead of an extension (G6) is that unnecessary dependencies are avoided; a class can be moved to a different package without losing its documentation. Finally, Fig. 7 depicts an ignore rule `IgnoreFileDocu` (no elements are created in the source domain), which states explicitly that adding documentation to an HTML file does not affect the source domain (G7). Note that this handles documentation for both packages and classes as the basic structure of the HTML file is identical.

Fig. 8 depicts an island `DocumentMethod`, a bridge `EOperationTableBridge`, and an ignore rule `Ignore-MethodDocu` used to handle creating methods and their corresponding documentation.

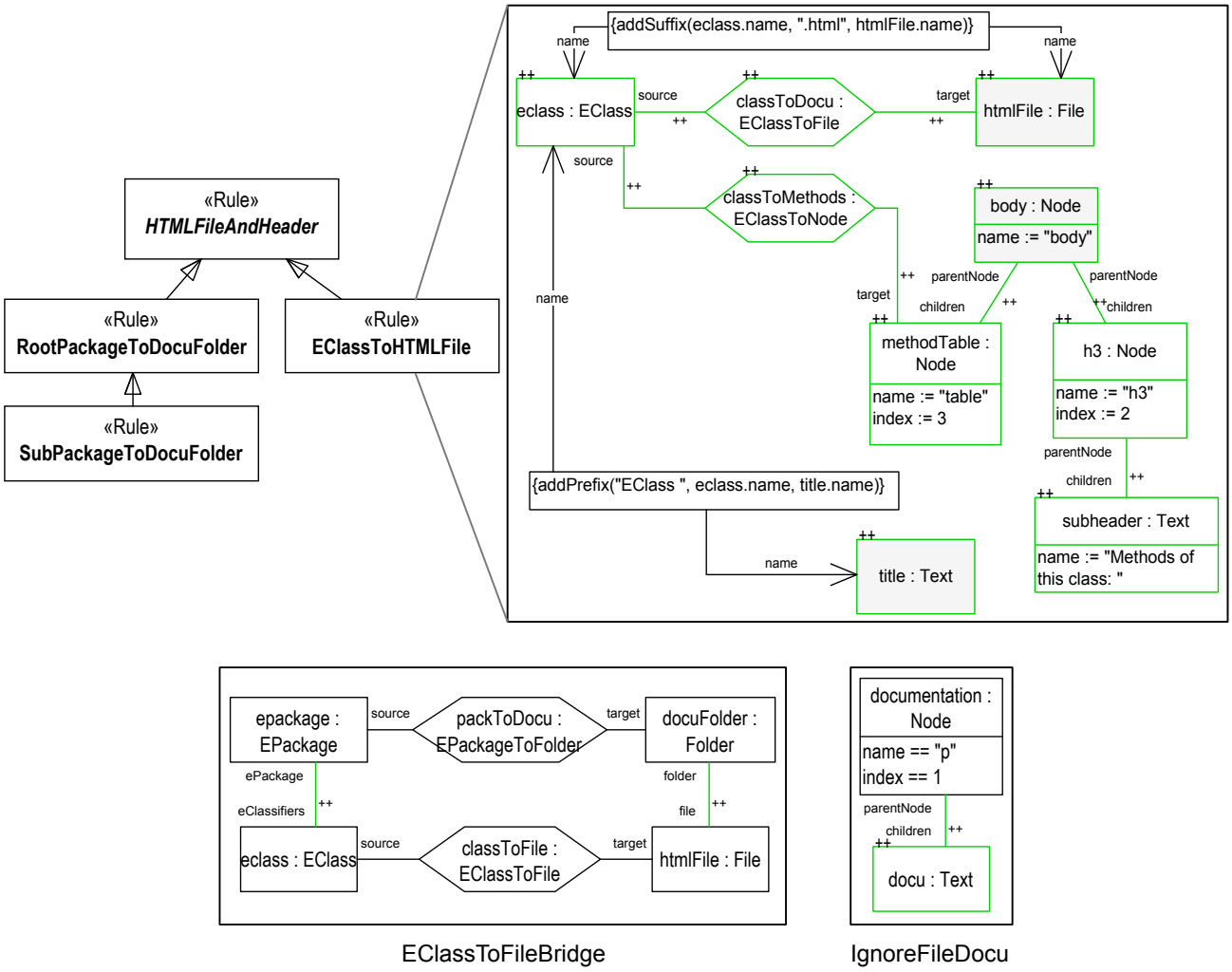

Figure 7: Handling classes and their documentation

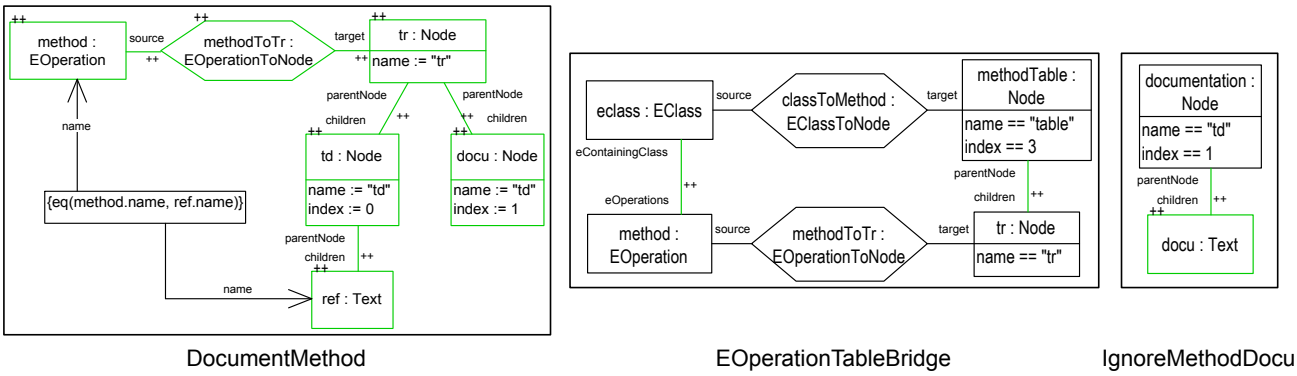

Figure 8: Handling methods and their documentation

Creating a method of a class corresponds to creating a row in an HTML table with two column entries: the first for the name of the method, and the second for its documentation (initially empty). Connecting a method to a class corresponds to adding this row to the method table of the class that was created by `EClassToHTMLFile`. A row in an HTML table is a `tr` element, while column entries are `td` elements. Indices are used in `DocumentMethod` to ensure that the column entries are always in the same order (name before documentation). `EOperationTableBridge` connects a method to a class, and adds its HTML row node to the method table of the class. Analogously to `IgnoreFileDocu`, the ignore rule `IgnoreMethodDocu` states that documenting a method does not affect the source domain. Note that the node `documentation` is constrained to be a column entry.

According to G7, we would actually need to specify ignore rules for all irrelevant concepts in the source domain (cf. Fig. 2) including inheritance relations, attributes, references, and datatypes. This guideline can be relaxed, however, for types that do not occur in *any* TGG rule. It is, for instance, easy to automatically ignore the reference `eSuperTypes` denoting inheritance, but difficult to automatically ignore `Text` documentation nodes (as in `IgnoreMethodDocu`), as the type `Text` does occur, e.g., as `ref:Text` in `DocumentMethod`.

3.4 Profile for hotspots and scalability

Declarative languages such as TGGs certainly have their advantages, but one must reserve some time for profiling and testing for scalability. Depending on the TGG tool and the underlying modelling framework and transformation engine, certain patterns and constructs can be, especially for beginners, surprisingly inefficient.

Guideline 8 (Use a profiler to test regularly for hotspots)**.** *Regularly profiling the synchronization for medium and large sized models is important to avoid bad design decisions early enough in the development process*[4]*.*

To support such a scalability analysis, some TGG tools [11, 23] provide support for generating models (of theoretically arbitrary size) using the TGG specification directly. This should be exploited if available.

Concrete optimization strategies for TGGs are discussed in detail in [15]. The most common user-related optimization is to provide more specific (and in some cases redundant) context elements and attribute constraints within the source and target domains of a rule. This helps the transformation engine eliminate inappropriate rule applications in an early phase, i.e., before checking all (costly) inter-model connections in a rule pattern.

4 TGG-Based Synchronization Algorithms

In this section we try to bring our TGG to life in the synchronization scenario depicted in Fig. 9. Our goal is to impart a high-level intuition for how TGG-based synchronization algorithms work in general. This provides not only a rationale for the guidelines already provided in the previous section, but also an understanding for how arbitrary deltas (and not only the explicitly specified creating deltas) are propagated. How this propagation works arguably depends on the chosen TGG tool, in our case eMoflon whose algorithm is described in [1], and we refer to [12, 16] for a detailed comparison of TGG tools. Nevertheless, the provided intuition is still helpful in understanding the TGG-related consequences of a delta, i.e., which TGG rule applications from former runs must be invalidated or preserved, and which TGG rules must be applied in a new run. In the following, the labels ① – ⑥ are used to refer to certain parts of the synchronization scenario in Fig. 9.

Batch Translation ①, ②: The scenario starts with the creation of a new class diagram ①. To provide a consequent delta-based intuition, this initial step can also be seen as a large source delta consisting of adding all elements in the class diagram. To create this class diagram ①, a root folder `TGGLanguage` containing two classes and two subfolders `compiler` and `precompiler` must be created. Each subfolder also contains a class, and `compiler` contains a subfolder as well. To propagate this source delta ②, TGG-based algorithms compute a sequence of TGG rule applications that would create the resulting source model. A possible sequence in this case is: (1) `RootPackageToDocuFolder` to create `TGGLanguage`, (2) `SubPackageToDocuFolder` applied three times to create `compiler`, `precompiler`, and `compilerfacade`, (3) `EClassToHTMLFile` to create all classes, and (4) `EClassFileBridge` to connect all classes to their containing packages. As TGG rules are specified as *triple* rules, applying this sequence of rules yields not only the expected source model but also a correspondence and a target model. In addition, the resulting triple of source, correspondence, and target models is, by definition, in the specified TGG language and is, therefore, *correct*. The target model is thus the result of forward propagating ② the source delta ①. Although this search for a possible rule application sequence can be implemented naïvely

[4] Although this ultimately depends on the specific metamodels and TGG rules, our current experience is that models of up to about 200 000 elements can be handled reasonably well by TGG-based synchronizers [15].

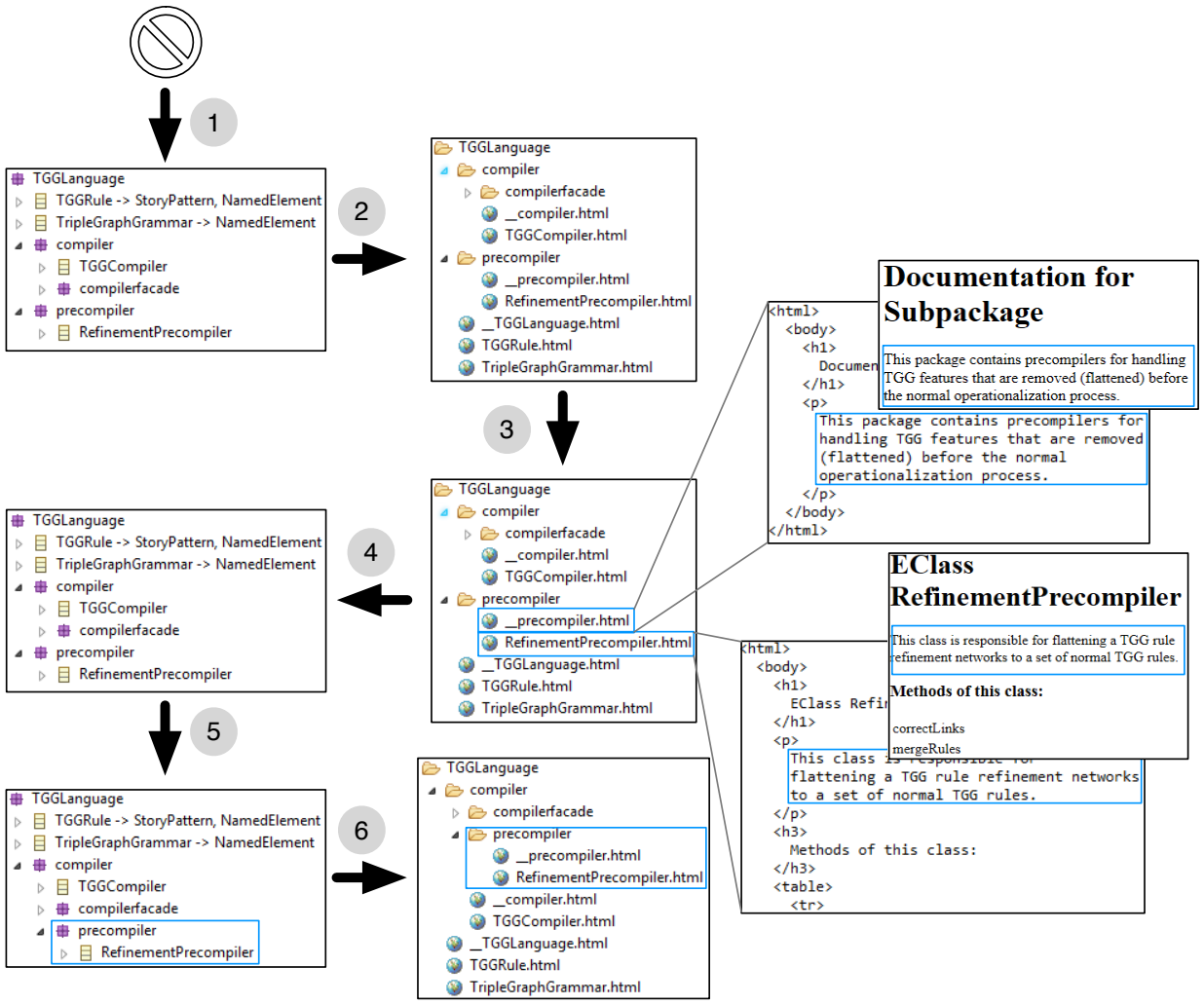

Figure 9: Synchronization scenario with our running example

as a straightforward trial and error procedure with backtracking, this would have exponential runtime and does not scale. All TGG tools we are aware of do not backtrack and instead restrict the class of possible TGGs.

A common restriction is demanding some variant of *confluence*, a well-known property of transition systems, which can be checked for statically using a so-called *critical pair analysis*. The interested reader is referred to, e.g., [2] for details. The naïve understanding of just fitting TGG rules until the correct sequence of rule applications is determined is, however, sufficient for an intuitive understanding of how this works in theory.

Ignoring Changes with Ignore Rules ③, ④: A target delta consisting of two changes is applied in ③. As depicted in Fig. 9, the documentation files `_precompiler.html` and `RefinementPrecompiler.html` for the package `precompiler` and the class `RefinementPrecompiler`, respectively, are edited. In terms of our HTML metamodel, this corresponds to adding a textual node to the corresponding paragraphs (`p` nodes) in the HTML files. A TGG-based synchronizer would propagate ④ this target delta by simply doing nothing, i.e., the source model is not changed at all. The same process is taken as with ②, i.e., a sequence of TGG rules is determined that applies the given target delta. The sequence in this case is: `IgnoreFileDocu` applied twice to create the added text node in both HTML files. As both rules are specified as *ignore* rules, the correspondence and target models are not changed in the process.

Handling Deleted and Added Elements ⑤, ⑥: After discussing two simple cases, we now demonstrate how the choice between extensions and bridges affects the behaviour of a TGG-based synchronizer. Let us consider a more general source delta ⑤, which "moves" the subpackage `precompiler` from `TGGLanguage` to `compiler`.

This is accomplished by (i) deleting the link between `TGGLanguage` and `precompiler`, and (ii) creating a new link between `compiler` and `precompiler`. Propagating this source delta ⑥ thus entails handling a deletion and an addition. The synchronization is, therefore, executed in three phases: a deletion phase, an addition phase, and a translation phase:

```
for all deletions:  revoke all dependent rule applications
for all additions:  revoke all dependent rule applications
translate all revoked and newly added elements
```

To *revoke* means to rollback a rule application, i.e., for forward synchronization, all correspondence and target elements created by the rule application to be revoked are deleted, while all created source elements are considered as revoked and to be (re)translated in the ensuing translation phase.

For every element that has been deleted, all rule applications that require the element as context and thus *depend* on the deleted element become invalid and must be revoked. Note that this must be done transitively.

In general, additions also have to be handled analogously to deletions, i.e., sometimes rule applications must be revoked as a consequence of newly added elements. For our running example, consider *adding* a new root package `eMoflonLanguages` that contains the current root package `TGGLanguage`. This must lead to revoking `TGGLanguage` and re-translating it as a subpackage of the new root package `eMoflonLanguages`!

To explain this further with our synchronization scenario, Fig. 10 depicts a relevant excerpt of the source and target models involved (top left). The deleted link is depicted bold and red with a "`--`" markup for emphasis. A TGG-based synchronizer typically keeps track of the translation by grouping links and objects into *rule applications*. This grouping is depicted visually in Fig. 10 for all rule applications required to create the current triple. For example, the rule application `3:SubPackageToDocuFolder` comprises the deleted link between `TGGLanguage` and `precompiler`, the subpackage `precompiler`, as well as the corresponding target elements: the link between the root folder `TGGLanguage` and subfolder `precompiler`, the folder `precompiler`, and the HTML file `__precompiler.html`. Note that the rule applications also comprise all relevant correspondence elements, which are abstracted from in Fig. 10 to simplify the explanation.

This grouping into rule applications, which can be reconstructed by simulating the creation of a consistent triple from scratch if necessary, is used to determine dependencies between the groups of elements. This is depicted visually in Fig. 10 (top right) as a dependency graph, showing that, for example, the rule application `5:EClassFileBridge` depends on (shown as an arrow) `3:SubPackageToDocuFolder` and `4:EClassToHTMLFile`, as these two rule applications create objects that are required as context in `5:EClassFileBridge`.

Given a source delta and calculated dependencies between rule applications, the deletion phase is carried out by revoking all rule application containing deleted elements, recursively revoking all rule applications that directly or transitively depend on revoked rule applications, and finally removing the deleted source elements from all data structures. This process is depicted in Fig. 10 (top right) showing (with a bold, red outline and "`--`" markup) that `3:SubPackageToDocuFolder` is to be revoked as it contains the deleted source link between `TGGLanguage` and `precompiler`. In this case, the only dependent rule application is `5:EClassFileBridge`, which must also be revoked (bottom right of Fig. 10).

In the translation phase, all revoked source elements are treated as if they were newly added, i.e., they are translated together with all added elements. This is depicted in Fig. 10 (bottom left) showing (bold, green outline and "`++`" markup) the revoked source elements `precompiler` and the link between `precompiler` and `RefinementPrecompiler`, as well as the newly added link between `compiler` and `precompiler` from the source delta. In this state, the same translation strategy as explained for ② and ④ can be applied, however, only for all added and revoked source elements. Due to this process, the documentation added to `RefinementPrecompiler.html` in ③ is retained as its corresponding class is not revoked (Fig. 10, bottom left). As the subpackage `precompiler` is, however, revoked and re-translated, its documentation file `__precompiler.html` is deleted, losing the changes made in ③, and is re-created afresh.

This is a direct consequence of using an extension rule for handling subpackages, but a bridge to connect classes to their parent packages. Similarly, the decision to use a bridge to connect methods to their classes enables, e.g., a *pull-up method* refactoring in the class diagram without having to revoke the method and lose its documentation. One can certainly argue that this behaviour is not always optimal, but it is what can currently be expected from state-of-the-art TGG-based synchronizers, given the choices we made when designing our TGG. Further improving current TGG synchronization algorithms to handle extensions as bridges during change propagation and still guarantee correctness is ongoing research.

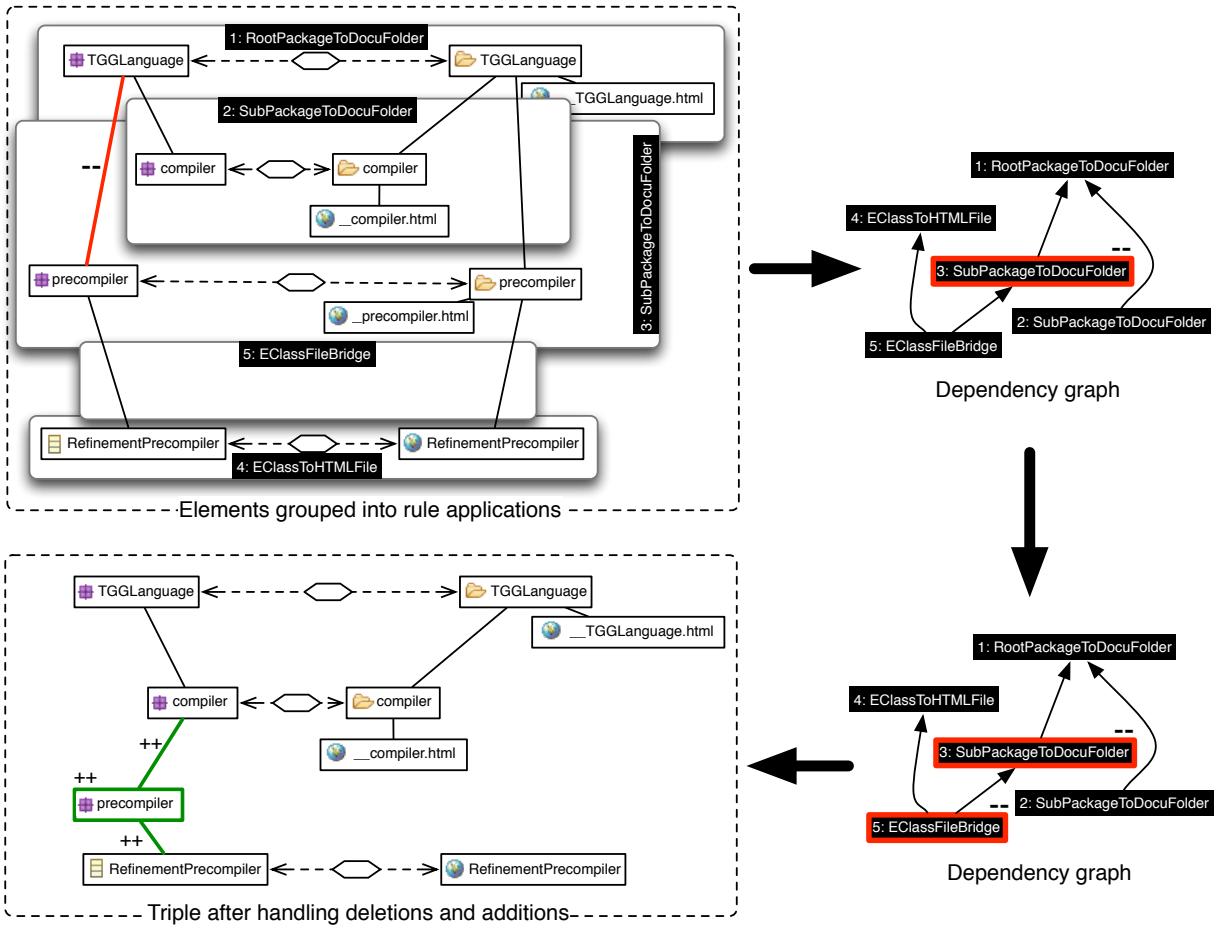

Figure 10: Handling of additions and deletions by a TGG-based synchronization algorithm

5 Related Work

We consider three different groups of related work in the following: (1) previous TGG publications with a focus similar to this work, (2) guidelines for *bx* approaches based on QVT-R which, similar to TGGs, address *bx* in an MDE context, and (3) other approaches to systematically developing model transformation rules in general.

Other approaches to systematically developing a TGG: Kindler and Wagner [13, 22] propose to semi-automatically synthesize TGG rules from exemplary pairs of consistent source and target models. Viewing their iterative synthesizing procedure through our geographical intuition, an island is first constructed from a given pair of consistent models, and is then stepwise deconstructed to smaller islands, bridges, or extensions by removing parts already covered by rules synthesized from former runs. As the procedure largely depends on the complexity of the transformation as well as the representativeness and conciseness of the provided examples, the authors suggest finalizing the process with manual modifications, for which our guidelines can be used.

Engineering guidelines are presented in [15] based on experience with profiling and optimizing TGGs. In contrast to this paper, the topics handled in [15] are mostly scalability-oriented and address, in many cases, primarily TGG tool developers rather than end users.

Guidelines for *bx* with QVT-R: Similar to TGGs, QVT-Relations (QVT-R) addresses *bx* in an MDE context. The standard QVT-R reference [17] already supplies examples demonstrating the usage of different language constructs. The standardized textual concrete syntax facilitates the distribution of such best practices across different QVT-R tools, whereas TGG tools currently suffer from interoperability issues as different metamodels are used for representing graphs, rules, and correspondences. Considering the chronological order of formal and practical contributions for TGGs and QVT-R, one can observe that while we now strive to impart a practical intuition and guidelines to complement existing formal work on TGGs, recent papers on QVT-R [4, 9, 20] strive to formalize concepts for an existing intuition.

Approaches to designing model transformation rules in general: Most of the work to facilitate transformation rule development focuses on semi-automatic usage of exemplary model pairs. While not necessarily focusing on *bx*, such approaches (e.g., [21, 24]) often require explicit mappings on the instance level, which closely resemble the correspondences in TGG rules. Further related contributions are those based on *design patterns* for model transformation [7, 10]. Although such design languages help to share solution strategies based on a common representation, our experience (especially with TGGs and QVT-R) is that the concrete choice of transformation language has a substantial impact on the proper way of thinking about a notion of consistency. In case of TGGs, for example, one must refrain from planning with control flow structures or (recursive) explicit rule invocations; features that are not necessarily excluded from other (*bx*) languages or general design pattern languages. Finally, this paper was inspired by the work of Zambon and Rensink in [25], where they demonstrate best practices for the transformation tool GROOVE using the N-Queens problem.

6 Conclusion and Future Work

In this paper, we have presented not only a basic introduction to TGGs, but also a process and a set of guidelines to support the systematic development of a TGG from a clear, but unformalised understanding of a *bx*.

We have, however, only been able to handle the basics and leave guidelines for advanced language features and techniques to future work including: negative application conditions, multi-amalgamation, user-defined attribute manipulation, test generation, and how best to specify the correspondence metamodel.

Acknowledgements. The first author of this paper received partial funding from the European Union's Seventh Framework Program (FP7/2007-2013) for CRYSTAL-Critical System Engineering Acceleration Joint Undertaking under grant agreement No 332830 and from Vinnova under DIARIENR 2012-04304.

References

[1] Anthony Anjorin. *Synchronization of Models on Different Abstraction Levels using Triple Graph Grammars*. Phd thesis, Technische Universität Darmstadt, 2014.

[2] Anthony Anjorin, Erhan Leblebici, Andy Schürr, and Gabriele Taentzer. A Static Analysis of Non-Confluent Triple Graph Grammars for Efficient Model Transformation. In Holger Giese and Barbara König, editors, *ICGT 2014*, volume 8571 of *LNCS*, pages 130–145. Springer, 2014.

[3] Anthony Anjorin, Karsten Saller, Malte Lochau, and Andy Schürr. Modularizing Triple Graph Grammars Using Rule Refinement. In Stefania Gnesi and Arend Rensink, editors, *FASE 2014*, volume 8411 of *LNCS*, pages 340–354. Springer, 2014.

[4] Julian C. Bradfield and Perdita Stevens. Recursive Checkonly QVT-R Transformations with General when and where Clauses via the Modal Mu Calculus. In Juan de Lara and Andrea Zisman, editors, *FASE 2012*, volume 7212 of *LNCS*, pages 194–208. Springer, 2012.

[5] James Cheney, James McKinna, Perdita Stevens, and Jeremy Gibbons. Towards a Repository of Bx Examples. In K. Selccuk Candan, Sihem Amer-Yahia, Nicole Schweikardt, Vassilis Christophides, and Vincent Leroy, editors, *BX 2014*, volume 1133 of *CEUR Workshop Proc.*, pages 87–91, 2014.

[6] Krzysztof Czarnecki, John Nathan Foster, Zhenjiang Hu, Ralf Lämmel, Andy Schürr, and James Terwilliger. Bidirectional Transformations: A Cross-Discipline Perspective. In Richard F. Paige, editor, *ICMT 2009*, volume 5563 of *LNCS*, pages 260–283. Springer, 2009.

[7] Hüseyin Ergin and Eugene Syriani. Towards a Language for Graph-Based Model Transformation Design Patterns. In Davide Di Ruscio and Dániel Varró, editors, *ICMT 2014*, volume 8568 of *LNCS*, pages 91–105. Springer, 2014.

[8] Joel Greenyer and Jan Rieke. Applying Advanced TGG Concepts for a Complex Transformation of Sequence Diagram Specifications to Timed Game Automata. In Andy Schürr, Dániel Varró, and Gergely Varró, editors, *AGTIVE 2011*, volume 7233 of *LNCS*, pages 222–237. Springer, 2012.

[9] Esther Guerra and Juan de Lara. An Algebraic Semantics for QVT-Relations Check-only Transformations. *Fundamentae Informatica*, 114(1):73–101, 2012.

[10] Esther Guerra, Juan de Lara, Dimitrios S. Kolovos, Richard F. Paige, and Osmar Marchi dos Santos. Engineering model transformations with transML. *SoSym*, 12(3):555–577, 2013.

[11] Stephan Hildebrandt, Leen Lambers, Holger Giese, Dominic Petrick, and Ingo Richter. Automatic Conformance Testing of Optimized Triple Graph Grammar Implementations. In Andy Schürr, Dániel Varró, and Gergely Varró, editors, *AGTIVE 2011*, volume 7233 of *LNCS*, pages 238–253. Springer, 2011.

[12] Stephan Hildebrandt, Leen Lambers, Holger Giese, Jan Rieke, Joel Greenyer, Wilhelm Schäfer, Marius Lauder, Anthony Anjorin, and Andy Schürr. A Survey of Triple Graph Grammar Tools. In Perdita Stevens and James Terwilliger, editors, *BX 2013*, volume 57 of *ECEASST*. EASST, 2013.

[13] Ekkart Kindler and Robert Wagner. Triple Graph Grammars : Concepts, Extensions, Implementations, and Application Scenarios. Technical report, Software Engineering Group, Department of Computer Science, University of Paderborn, 2007.

[14] Felix Klar, Alexander Königs, and Andy Schürr. Model Transformation in the Large. In Ivica Crnkovic and Antonia Bertolino, editors, *ESEC-FSE 2007*, pages 285–294. ACM, 2007.

[15] Erhan Leblebici, Anthony Anjorin, and Andy Schürr. A Catalogue of Optimization Techniques for Triple Graph Grammars. In Hans-Georg Fill, Dimitris Karagiannis, and Ulrich Reimer, editors, *Modellierung 2014*, volume 225 of *LNI*, pages 225–240. Gesellschaft für Informatik, 2014.

[16] Erhan Leblebici, Anthony Anjorin, Andy Schürr, Stephan Hildebrandt, Jan Rieke, and Joel Greenyer. A Comparison of Incremental Triple Graph Grammar Tools. In Frank Hermann and Stefan Sauer, editors, *GT-VMT 2014*, volume 67 of *ECEASST*. EASST, 2014.

[17] OMG. MOF2.0 query/view/transformation (QVT) version 1.2. OMG document formal/2015-02-01, 2015. Available from www.omg.org.

[18] Andy Schürr. Specification of Graph Translators with Triple Graph Grammars. In Ernst W. Mayr, Gunther Schmidt, and Gottfried Tinhofer, editors, *WG 1994*, volume 903 of *LNCS*, pages 151–163. Springer, 1994.

[19] Perdita Stevens. A Landscape of Bidirectional Model Transformations. In Ralf Lämmel, Joost Visser, and João Saraiva, editors, *GTTSE 2007*, volume 5235 of *LNCS*, pages 408–424. Springer, 2008.

[20] Perdita Stevens. A Simple Game-Theoretic Approach to Checkonly QVT Relations. *SoSym*, 12(1):175–199, 2013.

[21] Dániel Varró. Model Transformation by Example. In Oscar Nierstrasz, Jon Whittle, David Harel, and Gianna Reggio, editors, *MoDELS 2006*, volume 4199 of *LNCS*, pages 410–424. Springer, 2006.

[22] Robert Wagner. *Inkrementelle Modellsynchronisation*. PhD thesis, Universität Paderborn, 2009.

[23] Martin Wieber, Anthony Anjorin, and Andy Schürr. On the Usage of TGGs for Automated Model Transformation Testing. In Davide Di Ruscio and Dániel Varró, editors, *ICMT 2014*, volume 8568 of *LNCS*, pages 1–16. Springer, 2014.

[24] Manuel Wimmer, Michael Strommer, Horst Kargl, and Gerhard Kramler. Towards Model Transformation Generation By Example. In *HICSS-40 2007*, page 285. IEEE, 2007.

[25] Eduardo Zambon and Arend Rensink. Solving the N-Queens Problem with GROOVE - Towards a Compendium of Best Practices. In *GTVMT 2014*, volume 67 of *ECEASST*. EASST, 2014.