Existing and New Ideas on Least Change Triple Graph Grammars

Milica Stojkovic Sven Laux Anthony Anjorin

Paderborn University
{milica, slaux, aanjorin}@mail.uni-paderborn.de

Abstract

At least two actively developed model synchronization frameworks employ a conceptually similar algorithm based on Triple Graph Grammars as an underlying formalism. Although this algorithm exhibits acceptable behavior for many use cases, there are still scenarios in which it is sub-optimal, especially regarding the "least change" criterion, i.e., the extent to which models are changed to restore consistency. In this paper, we demonstrate this with a minimal example, discuss various suggestions from literature, and propose a novel approach that can be used to address this shortcoming in the future.

1 Introduction and Motivation

An important principle of the Model-Driven Software Development approach is to use suitable abstractions (models) to represent different aspects of a system. As such models are rarely isolated units and are often exposed to frequent modification, supporting and automating model consistency management via synchronization frameworks is an important task. For such frameworks to be trusted and used, a high quality of synchronization results must be guaranteed. A challenging quality criterion is the Least Change Principle [5, 7], which demands that no unnecessary (with respect to a concrete application scenario) modifications be made to restore consistency.

Triple Graph Grammars (TGGs) [8] are a rule-based formalism used to specify a consistency relation over two graph languages. There exist numerous TGG-based synchronization frameworks, including the framework of Hermann et al. [3], which is currently in use (with some variation) in at least two actively developed TGG tools. The synchronization algorithm proposed by Hermann et al. is, in our opinion, high-level, tool-agnostic, and conceptually simple as many (technical) details concerning, e.g., runtime complexity are omitted, making it advantageous for our equally high-level discussion. Although the algorithm exhibits acceptable behavior in general, there are still cases, especially concerning least change, where it is clearly sub-optimal.

To give an overview of the algorithm and explain the least change issues that can arise, we shall use a simple example concerning a consistency relation over folder structures in some file system, and a (UML) package hierarchy. The metamodels for the example are depicted in Fig. 1: On the left, the "source" metamodel defines a tree of named `Folder`s, on the right, the "target" metamodel defines a corresponding hierarchy of named `Package`s. To connect these two metamodels, the correspondence type `F2P` is defined as connecting corresponding pairs of `Folder`s and `Package`s. A triple of models that conforms to the triple of metamodels is depicted in Fig. 2.

Finally, Fig. 3 depicts two TGG rules: The first rule `R1` declares that a consistent pair of `Folder` and corresponding `Package` can always be established as root elements of a folder structure and package hierarchy by creating the green (++) elements in the rule. The second rule `R2` declares a structural pre-condition (black elements) that must be located (matched) in a host graph as required context, before the rule can be applied. `R2` specifies how subsequent `Folder`s and corresponding `Package`s can be added to extend an existing folder

Copyright © by the paper's authors. Copying permitted for private and academic purposes.

In: R. Eramo, M. Johnson (eds.): Proceedings of the Sixth International Workshop on Bidirectional Transformations (Bx 2017), Uppsala, Sweden, April 29, 2017, published at http://ceur-ws.org

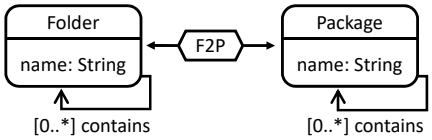

Figure 1: Metamodels

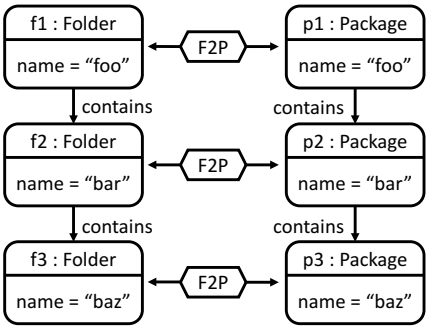

Figure 2: Exemplary model triple

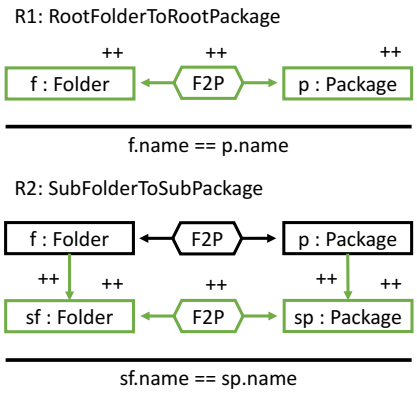

Figure 3: Rules `R1` and `R2`

structure and package hierarchy. The same attribute condition `f.name == p.name` is used in `R1` and `R2` to require identical names of corresponding `Folder`s and `Package`s. The model triple depicted in Fig. 2 is said to be consistent with respect to the TGG consisting of rules `R1` and `R2`, because it can be created by applying `R1` to the empty triple graph, then `R2`, and `R2` again as required.

Assume the folder structure is now updated by deleting `f2` and reconnecting `f1` directly with `f3`. To restore consistency, this change must be forward propagated to the package hierarchy. According to Hermann et al. [3], forward synchronization is performed by applying three auxiliary functions: Forward alignment (`fAln`), then deletion (`Del`) and finally forward addition (`fAdd`) (backward synchronization is analogous).

The task of `fAln` is simply to delete all “dangling” correspondence elements (referred to as “corrs”), i.e., all corrs that were previously connected to deleted source elements. As we deleted the folder `f2`, the corr connecting it to `p2` will be dangling and must be deleted as well. This situation is depicted in Fig. 4, showing the model triple after the source update but with the corr to be deleted highlighted in red (grey in a b/w printout).

The task of `Del` is to determine a maximal consistent sub-triple of the current state after applying `fAln`. To compute this maximal consistent sub-triple, `Del` attempts to mark the entire triple graph by applying suitable TGG rules. This marking process is visualized in Fig. 5 using checkboxes to represent the (un)marked state of all nodes and the edges that connect the individual triples accordingly (edges can be uniquely inferred for the example). The root elements `f1`, `p1`, and their connecting corr can be successfully marked with rule `R1` (successfully applied rules are denoted in square brackets). The remaining elements cannot be marked with any rule, however, and are left unmarked. When the marking process terminates, `Del` deletes all unmarked elements (highlighted in Fig. 5 in red/grey) in the correspondence and target models.

In a final step, the third function `fAdd` applies TGG rules to transform all unmarked elements in the source model, extending the correspondence and target models with new elements in the process. The final consistent state after applying `fAdd` is depicted in Fig. 6, with the newly created elements highlighted in green (grey).

While the algorithm is correct, complete, stable, and invertible (cf. [3] for all details), our simple example shows clearly that it makes a rather rough approximation of the elements that must be deleted (and recreated) to ensure consistency and thus performs unnecessary changes. For a model triple with a very deep folder structure

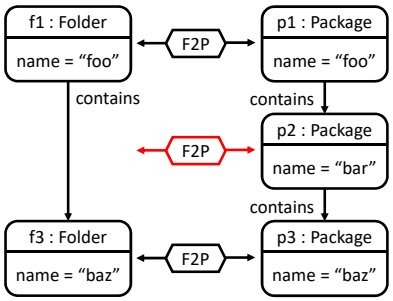

Figure 4: Application of `fAln`

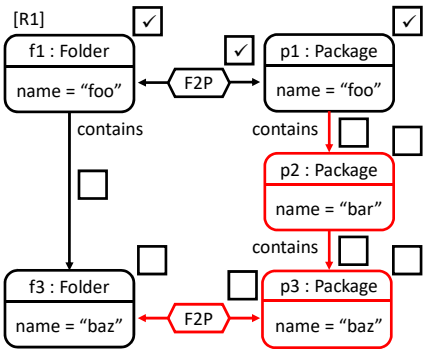

Figure 5: Application of `Del`

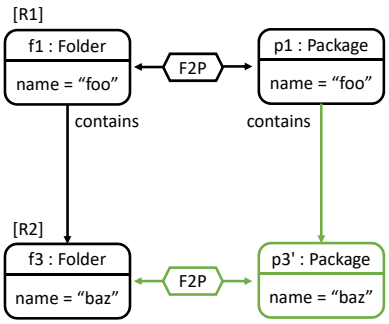

Figure 6: Application of `fAdd`

and corresponding package hierarchy, removing the second folder as we did would result in the deletion and identical re-creation of *all* sub-packages. For a realistic example, this will incur substantial information loss, as the sub-packages will probably have attribute values and other contents such as files with manually written code, which cannot be re-generated from the source model. Even without rigorously formalizing least change, this is clearly sub-optimal and most probably unwanted synchronization behavior.

2 Existing Ideas on How to Improve Least Change Behavior for TGGs

An approach to improve the TGG algorithm is suggested in [2]. The idea is not to delete elements right away with `Del`, but rather to mark them for deletion and try to reuse them later with `fAdd` instead of re-creating them. This breaks the clear separation of `fAln`, `Del` and `fAdd`, allowing the functions to work together during the synchronization process. Applying this idea to our example, the improved algorithm would mark the target element `p2` and its connected corr for deletion *without* actually deleting any of those elements (resulting in a "lazy" `fAln+Del`). During translation, before actually (re-)creating any new elements, `fAdd` can now first check in the "deletion pool" if a suitable element can be recycled. In this manner, the package `p3` and its corr to the folder `f3` would only be marked for deletion in Fig. 5 and then re-used (instead of being re-created as before). The process is finalized by deleting all elements remaining in the deletion pool, in this case the package `p2` and its dangling corr. While this can potentially avoid information loss by recycling elements, choosing the "right" elements to re-use makes the whole process highly non-deterministic and could either overwhelm end-users if this freedom of choice is left open, or surprise them if the decision is internalized and automated.

A different approach proposed by Hildebrandt [4] is to specify or automatically derive additional "fix rules" for the sole purpose of restoring consistency locally. This idea can be applied to the algorithm by Hermann et al. [3] by replacing `fAln` with a function `fix`, that attempts to restore consistency locally. Such fix strategies can potentially enlarge the maximal consistent sub-triple determined subsequently by `Del`. In our example, such a local `fix` would be to mirror the update in the target model by deleting `p2` (with its dangling corr) and reconnecting `p1` with `p3`. In this case, `Del` would be able to mark the entire triple, avoiding unnecessary deletion and re-creation. It is, however, unclear how to derive these fix rules automatically from arbitrarily complex TGGs, or if they are manually specified, how to guarantee that they do not contradict the original TGG.

Finally, practical guidelines for developing a TGG are presented by Anjorin et al. [1], including a "best practice" guideline concerning least change. Inspecting rule `R2` in Fig. 3, we can see that sub-packages depend on their parent package as context. This means that the existence of the parent package is a precondition for the existence of the sub-package. For our example, however, this context dependency is *not* absolutely necessary; Anjorin et al. explain that dependencies like this should be avoided by splitting such rules into an "island rule" that only creates elements (a folder and corresponding package) and another "bridge rule" that adds the required edges (connecting the folder and package to their parents). Applying this guideline to our example replaces the "extension rule" `R2` in Fig. 3, with the bridge `R2'` depicted in Fig. 7. According to the least change guideline [1], bridges are better than extensions as they reflect the actual context dependencies more accurately. With this "improved" TGG, synchronization is performed as follows: `fAln` behaves the same as in Fig. 4; when computing the maximal consistent sub-triple, however, `Del` can now mark the elements below the deleted folder `f2` using rule `R1` (Fig. 8). This means that only the package `p2` and its incident edges are deleted. After deleting these elements, `fAdd` can now translate the unmarked edge connecting `f1` and `f3` with the bridge rule `R2'` (Fig. 9).

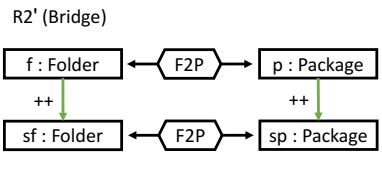

Figure 7: Bridge rule `R2'`

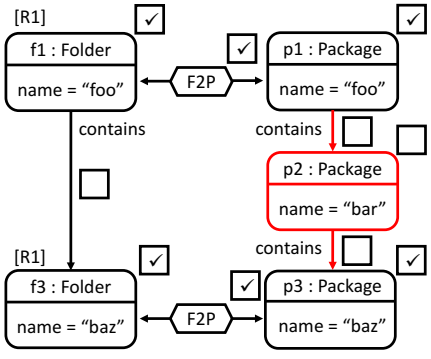

Figure 8: After application of `Del`

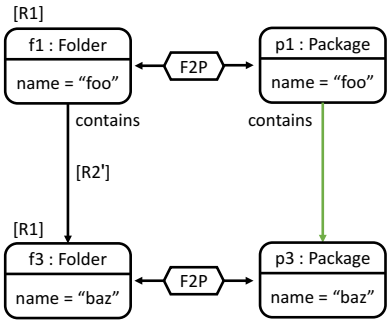

Figure 9: After application of `fAdd`

Although this works for our example, there are two issues: Firstly, replacing extension with bridge (and island) rules can *change* the language generated by the TGG, resulting in synchronizers that now accept and propagate previously invalid updates to produce results that are inconsistent with respect to the original TGG. As an example, note that it becomes possible to create arbitrary *graphs* of folders and packages using the bridge rule `R2'`, while the original extension rule `R2` could only generate trees. In this case, suitable application conditions for `R2'` could be used to solve the problem, but this requires challenging, manual effort with no guarantees that both TGGs are equivalent. Secondly, although the refactored TGG might be more suitable for synchronization, TGGs are used for other consistency management tasks where this change might have a negative impact, e.g., consistency checks or model generation. The TGG developer might also prefer a certain style of specification, e.g., for readability reasons, and should not be forced to adhere to a certain structure for rather technical reasons.

3 Combining Optimization Techniques and TGGs to Improve Least Change

Our idea stems mainly from the observation that `Del` establishes a maximal consistent sub-triple, which *must* be in the language generated by the TGG. For large TGG rules (e.g., extension instead of island and bridge rules), this leads to coarse, rigid steps in the synchronization process. The algorithm can thus be improved by automatically decomposing a given TGG into a new, more flexible TGG consisting of rules that are as small as possible. While this enables taking smaller, more precise steps in the synchronization process, appropriate constraints ranging over valid rule application sequences must be automatically generated to ensure that the final result is again in the language of the original TGG. To demonstrate why such constraints are required, we now discuss the forward propagation of an *inconsistent* update: instead of deleting the folder `f2`, let us assume that our file system allows us to add a symbolic link between folder `f1` and folder `f3` as depicted in Fig. 10, but that a corresponding modification is *not* allowed in the language of packages.

A tool based on our original TGG would refuse to forward propagate this update, as there is no rule that can create edges without creating the corresponding sub-folders and sub-packages. Using the decomposed bridge rule `R2'`, however, the invalid change would be propagated to the target model, resulting either in an inconsistent triple or, perhaps worse, in a runtime exception as the target model cannot be manipulated in this manner.

In order to correctly detect and reject this inconsistent update, we will use the following four steps:

1. Decompose the original TGG rules to derive a new TGG and corresponding constraint derivation logic.
2. Start the synchronization process using the decomposed rules and collect all possible rule applications.
3. Generate specific constraints based on the constraint derivation logic from Step 1.
4. Solve the constraint problem to make a guaranteed consistent choice of rule applications from Step 2.

The first step is to decompose complex rules into smaller ones. Extension rules can, for example, be split into island and bridge rules, which can themselves be decomposed further if they create multiple elements. For our example, extension rule `R2` can be decomposed to produce the bridge rule `R2'` depicted in Fig. 7. To ensure that the language of the TGG is not changed by this decomposition, constraints demanding that every application of `R2'` be exclusively paired with a suitable application of `R1` (creating `sf` and `sp`) must be derived during the synchronization process to ensure that a decomposed sub-derivation $\overset{R1}{\Longrightarrow}\overset{R2'}{\Longrightarrow}$ can be fused to an equivalent derivation $\overset{R2}{\Longrightarrow}$ with the original TGG rule. This constraint derivation logic is produced during the decomposition and stored for later use during synchronization.

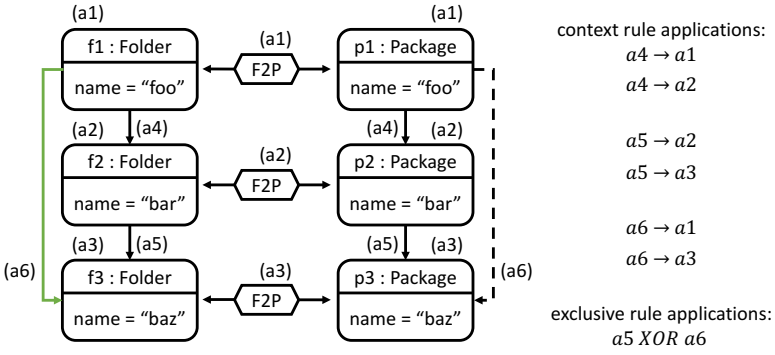

Figure 10: Performing synchronization with decomposed TGG rules

The next step concerns the synchronization process: Once the change has been applied, the decomposed language is used to collect all existing and potential rule applications. As depicted in Fig. 10, we can annotate each existing rule application by introducing corresponding variables `a1` to `a5`. Furthermore, we also annotate the *potential* rule application `a6`. Note that instead of performing any propagation directly, we only collect all existing and potential rule applications before considering additional constraints to guide our final choice.

Constraints are now generated based on the constraint derivation logic obtained from the decomposition in Step 1. For our example, we have to make sure that every time `R2'` is used to mark elements, there are complementary applications of `R1` marking the required context elements. To accomplish this there are two constraints that have to be taken into account: (1) context rule application constraints ensure that context requirements are fulfilled (e.g., `a4` depends on `a1` and `a2`), while (2) exclusive rule application constraints ensure that the elements are marked only once. This ensures here that only trees and not graphs can be created. In our example, both `a5` and the potential rule application `a6` depend on `a3`, but our constraint derivation logic demands that `a3` can only be combined with one of them, hence the derived constraint `a5 XOR a6`.

In a final step, consistency is ensured by solving an optimization problem over the set of constraints generated in Step 3. To influence this step, different objective functions can be specified depending on the desired outcome. For our example it would be reasonable to enforce least change by maximizing the number of marked elements, but in other scenarios one could prefer most recent changes, or ask the user to decide. For the situation depicted in Fig. 10, there exists no solution for which all elements are marked. Possible optimal solutions include rejecting the update (analogously to synchronization with the original TGG), and suggesting that `a5` be revoked instead.

The underlying idea of combining standard optimization techniques (e.g., SAT or ILP solvers) with TGGs has already been proposed by Leblebici et al. [6], and promising first results for the task of consistency checking have been obtained. Our suggestion for future work is to apply this technique to the task of synchronization as a means of improving the least change related quality of current TGG-based synchronizers.

Our suggestion generalizes and automates the island/bridge guideline, solving the two current problems as follows: Appropriate constraints are automatically derived to guarantee that the final result is again consistent with respect to the original TGG thus allowing intermediate, potentially inconsistent states and the flexibility for least change optimization. The TGG specified by the user remains unchanged as the optimal form for synchronization is derived automatically as part of the compilation process.

References

[1] Anthony Anjorin, Erhan Leblebici, Roland Kluge, Andy Schürr, and Perdita Stevens. A Systematic Approach and Guidelines to Developing a Triple Graph Grammar. In Alcino Cunha and Ekkart Kindler, editors, *BX 2015*, volume 1396, pages 81–95. CEUR-WS.org, 2015.

[2] Joel Greenyer, Sebastian Pook, and Jan Rieke. Preventing Information Loss in Incremental Model Synchronization by Reusing Elements. In Robert B. France, Jochen M. Kuester, Behzad Bordbar, and Richard F. Paige, editors, *ECMFA 2011*, volume 6698, pages 144–159. Springer, 2011.

[3] Frank Hermann, Hartmut Ehrig, Fernando Orejas, Krzysztof Czarnecki, Zinovy Diskin, Yingfei Xiong, Susann Gottmann, and Thomas Engel. Model Synchronization Based on Triple Graph Grammars: Correctness, Completeness and Invertibility. *Software and Systems Modeling*, 14(1):241–269, 2015.

[4] Stephan Hildebrandt. *On the Performance and Conformance of Triple Graph Grammar Implementations*. PhD thesis, University of Potsdam, 2014.

[5] Cheney James, Jeremy Gibbons, James McKinna, and Perdita Stevens. Towards a Principle of Least Surprise for Bidirectional Transformations. In Alcino Cunha and Ekkart Kindler, editors, *BX 2015*, volume 1396, pages 66–80. CEUR-WS.org, 2015.

[6] Erhan Leblebici. Towards a Graph Grammar-Based Approach to Inter-Model Consistency Checks with Traceability Support. In Anthony Anjorin and Jeremy Gibbons, editors, *BX 2016*, volume 1571, pages 35–39. CEUR-WS.org, 2016.

[7] Lambert Meertens. Designing Constraint Maintainers for User Interaction. Technical report, 1998.

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