Triple Graph Grammars or Triple Graph Transformation Systems?

A Case Study from Software Configuration Management

Thomas Buchmann, Alexander Dotor, and Bernhard Westfechtel

Angewandte Informatik 1, Universität Bayreuth
D-95440 Bayreuth
firstname.lastname@uni-bayreuth.de

Abstract. Triple graph grammars have been used to specify consistency maintenance between inter-dependent and evolving models at a high level of abstraction. On a lower level, consistency maintenance may be specified by a triple graph transformation system, which takes care of all operational details required for executing consistency maintenance operations. We present a case study from software configuration management in which we decided to hand-craft a triple graph transformation system rather than to generate it from a triple graph grammar. The case study demonstrates some limitations concerning the kinds of consistency maintenance problems which can be handled by triple graph grammars.

1 Introduction

Model transformations play a key role in model-driven engineering. In the most simple case, a transformation of some source model s into some target model t may be performed automatically by a model compiler. If there is no need to edit t, model evolution (of s) may be handled by simply compiling s once more. However, in general it may not be possible to generate t from s completely automatically, both s and t may evolve, and changes may have to be propagated in both directions (round-trip engineering).

Many formalisms have been proposed for defining model transformations, including e.g. QVT [1] in the context of object-oriented modeling. In this paper, we focus on *graph transformations*: Graphs may represent models in a natural way; graph transformation rules describe modifications of graph structures in a declarative way. Furthermore, there is a comprehensive body of theories, languages, tools, and applications (see e.g. the handbooks on graph grammars [2,3]).

A *directed, typed, attributed graph* consists of typed nodes which are decorated with attributes and are connected by typed, directed, binary edges. In terms of object-oriented modeling, a node corresponds to an object, and an edge corresponds to a binary link with distinguishable ends. A *graph grammar* is composed of a graph schema, which defines types of nodes, edges, and attributes, a start graph, and a set of *productions* which are used to generate a set of graphs forming a graph language. Thus, a graph grammar is concerned only with the generation of graphs. In contrast, a *graph transformation system* is made up of a set of *graph transformation rules*, which describe arbitrary graph transformations including deletions and modifications.

M.R.V. Chaudron (Ed.): MODELS 2008 Workshops, LNCS 5421, pp. 138–150, 2009.

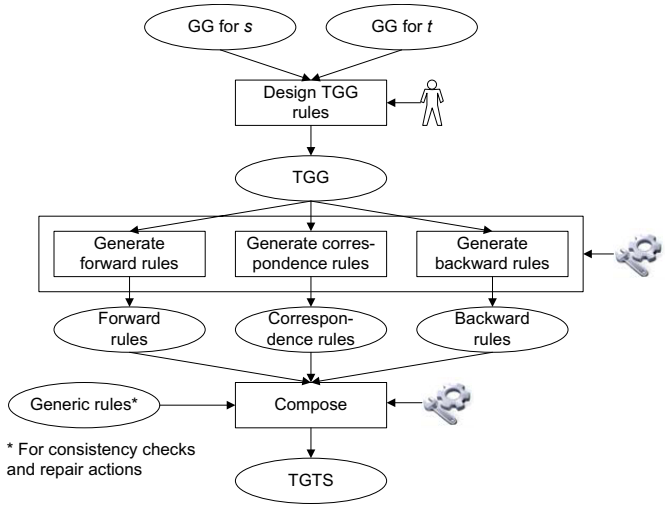


Fig. 1. The TGG approach

For maintaining consistency between interdependent and evolving models, a graph transformation system is required which deals with at least two graphs, namely the *source graph* s and the *target graph* t. For incremental change propagation, a *correspondence graph* c is placed in between s and t for maintaining *traceability links* with the help of *link nodes*. This results in a *triple graph transformation system (TGTS)*, the rules of which define source-to-target and target-to-source transformations as well as actions for checking consistency and repairing inconsistencies.

Developing a TGTS is still a tedious and laborious task: First, in addition to the generation of graphs, modifications and deletions have to be specified explicitly. Second, each direction of transformation has to be specified explicitly - as well as rules for checking consistency and establishing correspondences. Therefore, *triple graph grammars (TGG)* [4,5] have been proposed to leverage the specification of inter-model consistency maintenance. From a high-level TGG dealing only with the synchronous generation of graphs, a more low-level TGTS may be generated (Figure 1): The TGG designer need only define generative *triple rules* which describe synchronous extensions of source, target, and correspondence graph. From each synchronous triple rule, three *directed rules* may be generated (if required): A *forward rule* assumes that s has been extended, and extends c and t accordingly; likewise for *backward rules*. A *correspondence rule* extends c after s and t have been extended with corresponding graph structures. Like triple rules, directed rules are *monotonic*, i.e., they describe graph extensions. Directed rules are required when s, t, and c have not been changed synchronously (e.g., when different users edit s and t independently). In addition to directed rules, further rules are needed which deal with modifications and deletions having been performed in s and t. To this end, *generic rules* have to be defined which perform *consistency checks* and *repair actions*. Finally, the TGTS may include a (generic) control structure for efficient graph traversal to speed up the execution of consistency maintenance operations.

In this paper, we explore the alternatives of *hand-crafting a TGTS* versus defining a TGG and *generating a TGTS* from the TGG. To this end, we present a *case study* which we performed in the *MOD2-SCM* project (*MOD*el-Driven *MOD*ular *S*oftware *C*onfiguration *M*anagement System [6]), which aims at developing a model-driven product line for SCM systems. The project employs Fujaba [7] as modeling language and tool, but the discussion in this paper is not specific to Fujaba.

2 Case Study

In the context of developing SCM systems, a recurring problem to be solved consists in the *synchronization* between *repositories* and *workspaces*, which requires *bidirectional* and *incremental change propagation*. The case study was inspired by open source SCM systems such as Subversion or CVS. In the model we built, both files and directories are versioned in a uniform way. The version history of each file system object is represented by a *version tree*. A version of a directory uniquely determines the versions of its components. While file system objects are organized into strict hierarchies (trees), the hierarchy of file system object versions allows for sharing.

Synchronization between repositories and workspaces is supported by the following commands which, when applied to directories, are propagated to all components: add prepares a file system object for a commit into the repository. commit commits a file system object into the repository. For an added file system object, an initial version is created in the repository. For a changed versioned file system object, a successor version is created. checkout creates a fresh copy of some file system object version in the workspace. update propagates changes from the repository into the workspace. Unmodified copies of outdated versions are replaced with copies of new versions; modified copies are not updated to prevent the loss of local changes. Finally, checkStatus analyzes the consistency between repository and workspace. File system objects are marked as created, deleted, modified, moved, or out of date.

An example is given in Figure 2. The scenario starts with a repository which is initially empty and a small file system hierarchy (a). In step 1, add is applied to the hierarchy rooted at d1. All objects (files or directories) of this hierarchy are scheduled for commit. Adding is implemented by creating *link objects* and attaching them to the file system objects. In step 2, commit is used to import the file hierarchy into the repository. For each file system object, both a versioned object and its initial version are created. Furthermore, the hierarchy is mirrored in the repository both at the object and the version level. In step 3, a part of the hierarchy (with root d2) is exported into another workspace with checkout. In ws2, the text content of f2 is updated, and the name of f3 is changed into f4. Finally, in step 4 the changes in ws2 are committed. This causes new versions of both files to be created. Please note that the file names of different versions may differ. Furthermore, a new version of the directory d2 is created which includes the new versions of the changed files. An update propagates the changes from the repository to ws1; all files and directories now refer to the current versions. Please note that due to the lack of space we have not shown *structural variations* caused by moves and deletes.

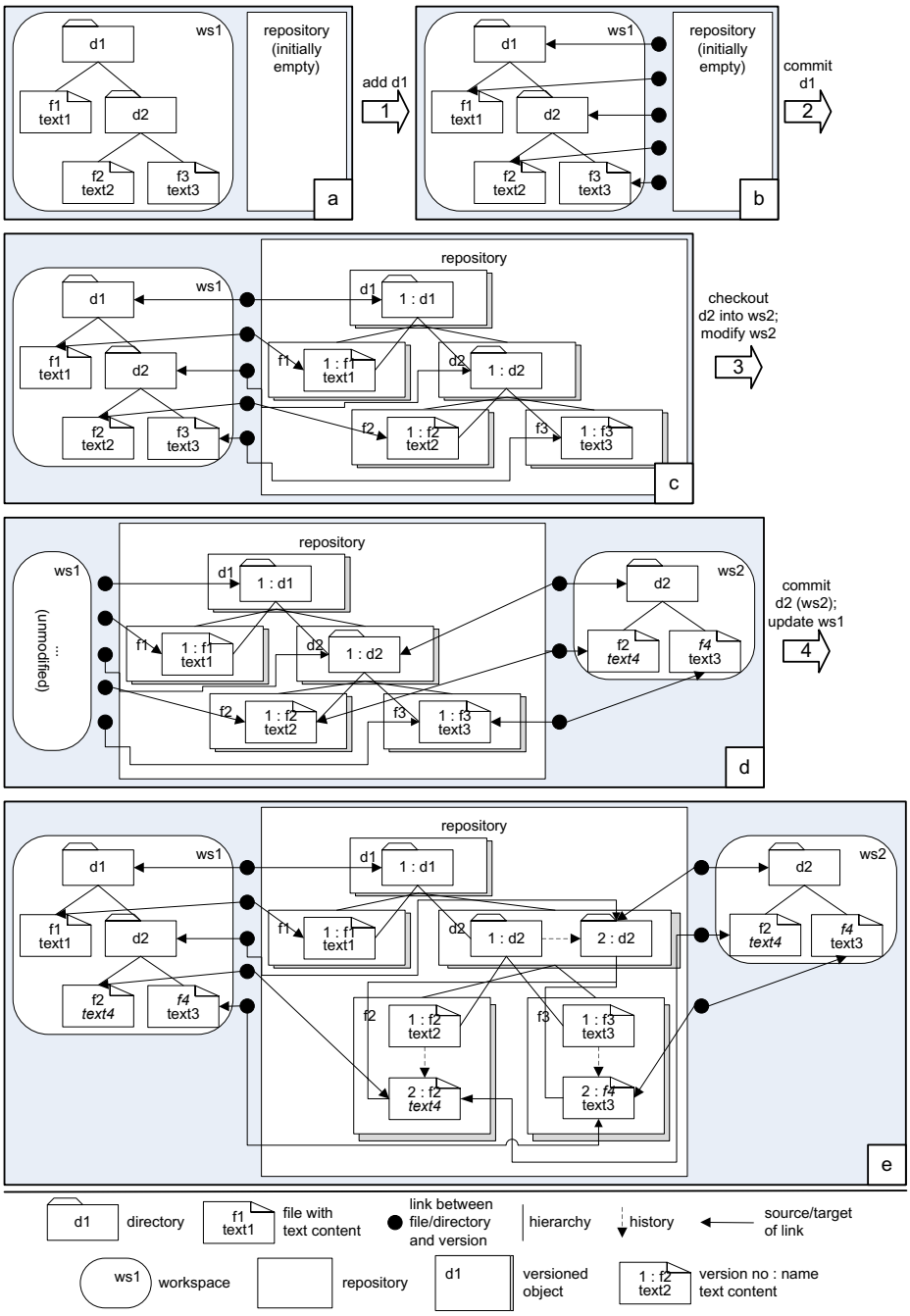


Fig. 2. Example

3 A Triple Graph Transformation System

In this section, we present the approach which we decided to follow in our case study. Synchronization of repositories and workspaces is modeled with the help of a *hand-crafted TGTS*. Generating a TGTS from a TGG is discussed in the next section.

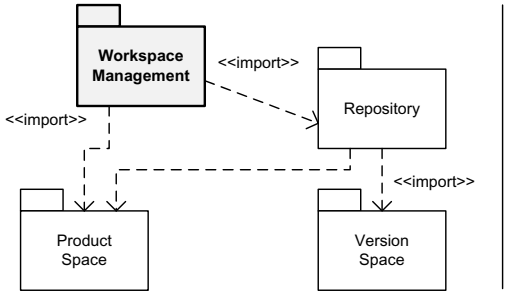
Modeling Language. The TGTS was created with the help of the object-oriented modeling language and environment *Fujaba* [7]. In Fujaba, nodes and edges are represented as objects and links, respectively. Types of nodes and edges are defined by *class diagrams*. The behavior of objects and links may be modeled by story patterns and story diagrams (see below). Models written in Fujaba are executable (compiled into Java code).

Story patterns are UML-like communication diagrams which can be used to represent graph transformation rules. A story pattern consists of a graph of objects and links. A graph transformation rule is expressed by marking graph elements as deleted or created. Furthermore, conditions may be defined for attribute values, and the values of attributes may be modified. Finally, methods may be called on objects of the story patterns. Thus, story patterns constitute a uniform language construct for defining graph queries, graph transformations, and method calls.

Programming with story patterns is supported by *story diagrams*, which correspond to interaction overview diagrams in UML 2.0. A story diagram consists of story patterns which are arranged into a control flow graph. Story patterns appear in-line in nodes of the control flow graph. Each story diagram is used to implement a method defined in the class diagram.

Model Architecture. Figure 3 shows a *package diagram* on the left and lists data on the overall model size on the right. Each *package* contains one class diagram and a set of story diagrams for the methods introduced in the classes of the package. A dashed line labeled with import denotes an *import* relationship. An unlabeled dashed line represents a *dependency* (the dependent package may have to be modified if the target package is changed).

Using the terminology of [8], we distinguish between a *product space*, which is composed of the elements to be submitted to version control, and a *version space*, where



number of classes and interfaces	24
number of attributes	27
number of methods	83
lines of generated Java code	13291

Fig. 3. Package diagram (left) and model size (right)

the evolution of these elements is represented. In the context of this paper, the product space consists of the file system hierarchy. The package VersionSpace introduces classes for managing the evolution of versioned objects. In the version model of the case study, the versions of a versioned object are organized into a history tree. The package Repository provides classes for versioned files and directories, based on the packages ProductSpace and VersionSpace. Composition hierarchies are defined on both object and version level. The packages mentioned so far will not be discussed further. Rather, we will focus on the package WorkspaceManagement.

Class diagram. Figure 4 shows the class diagram for the package Workspace Management. Classes imported from other packages are displayed with collapsed attributes and methods sections (left-hand and right-hand side). The class WorkspaceManager provides the external interface of methods for synchronizing repositories and workspaces (facade pattern).

Synchronization between repositories and workspaces is realized with the help of a *correspondence graph*, which is composed of *link objects* being connected to one source object and one target version, respectively. The abstract class Link provides a set of methods corresponding to the exported methods of WorkspaceManager, and a set of auxiliary methods. Furthermore, each link object carries a set of boolean attributes indicating the state of the link: In state created, the target of the link does not yet exist. In state deleted, the source of the link has been destroyed. In state modified, both source and target exist, and the source has been modified in the workspace. In state moved, the source has been moved to a different location. In state outOfDate, a new version of the target is available. Finally, in state updated, the source has been updated to a new version of the target, but this change has not been committed yet at the next higher level. Please note that these attributes are not mutually exclusive (e.g., a link object can be both out of date and modified).

Link objects are organized into a composition hierarchy in a similar way as file system objects (composite pattern). When the workspace is consistent with the repository, the composition tree for link objects agrees with the composition tree of file system objects in the workspace. The algorithms for synchronizing repositories and workspaces traverse the composition hierarchy. Since they are attached to the class Link and its subclasses, the classes of the imported packages need not be extended or modified.

Operations. To illustrate the style of modeling adopted in the TGTS, we present three simple examples of methods for synchronizing repositories and workspaces. All sample methods are attached to the class Link and perform only those parts of the synchronization which can be handled at this general level. The methods are redefined in the subclasses; propagation through the hierarchy is performed in the class CompositeLink. Only link objects and their embeddings are manipulated directly. All changes in the repository and the workspace are affected by method calls only.

Figure 5a shows the story diagram for committing the creation of a new file system object. The first story pattern checks the state of the link object and locates the root of the repository via the workspace manager. In the second step, a new versioned object is created. The third step creates the root version of this object, sets its name and its owner, and connects the new version to the link object.

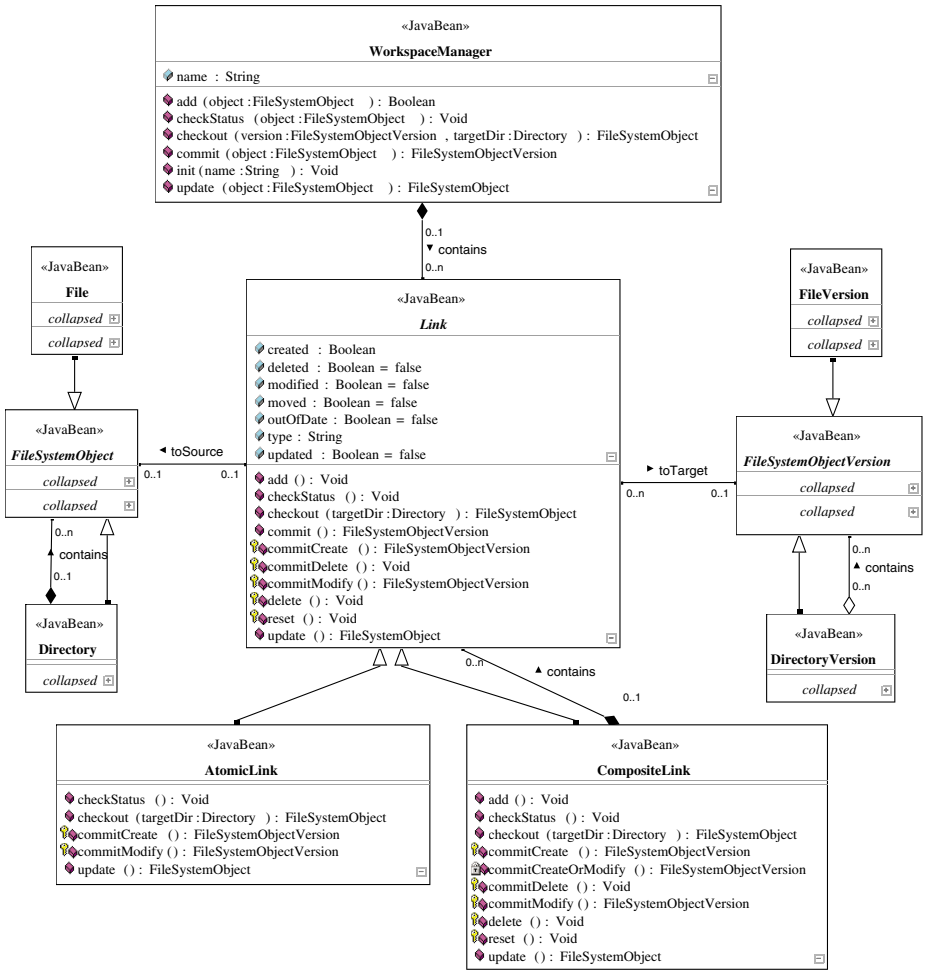
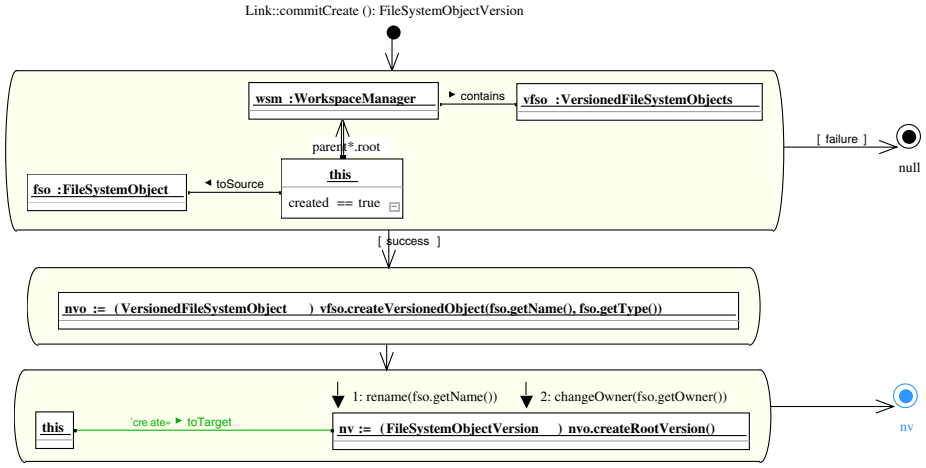


Fig. 4. Class diagram for the package Workspace Management

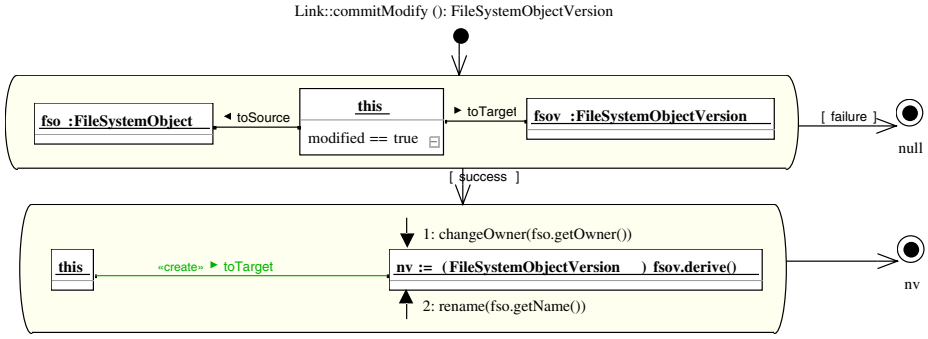
The story diagram of Figure 5b is invoked when a file system object is already under version control and has been modified in the workspace. In the first step, the state of the link object is checked, and both ends are retrieved. In the second step, a successor version of the old target is created, and the link object is redirected to the new target.

The story pattern of Figure 5c handles change propagation from the repository into the workspace. The link object must be both outOfDate and not modified; the latter condition prevents that changes in the workspace are overwritten inadvertently. The link object is redirected to refer to the most recent (transitive) successor of the old target. This is ensured by the negative application condition (crossed object): There must be no other successor with a higher version number. In the course of the update, the name and the owner of the file system object are copied from the new target version.

a)



b)



c)

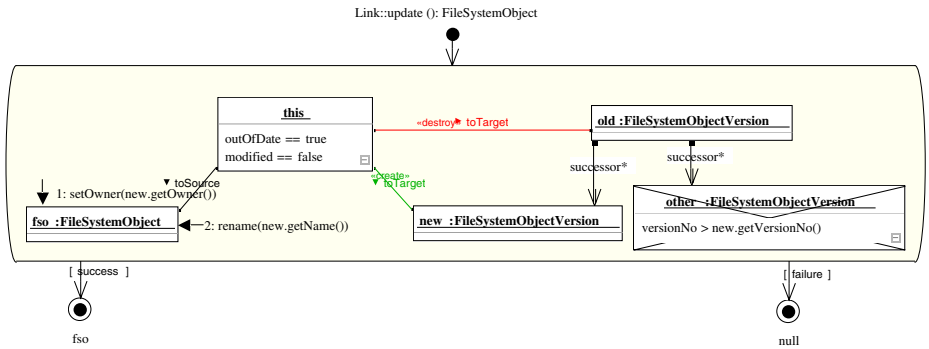


Fig. 5. Story diagrams for commits and updates

4 A Triple Graph Grammar?

This section discusses the application of the TGG approach to our case study. To this end, we modeled some sample TGG rules in Fujaba. In the presentation below, we describe several conceptual problems which we encountered.

Synchronous Rules. The first step of the TGG approach — defining a set of triple rules — is illustrated in Figure 6. The figure displays three rules which handle the creation of a subdirectory. Each rule is applied to a complex link both ends of which are already present. All rules perform the same extensions on the source graph and the correspondence graph: A subdirectory is created if there is no file system object of the same name; furthermore, a sublink is created and connected to the new subdirectory. The rules differ with respect to the target graph. The rule set is designed in such a way that all structures which may occur in the repository can be generated (version trees for the evolution history, acyclic graphs for the composition hierarchy).

The first rule creates a directory in the workspace along with a versioned object and an initial version in the workspace. The attribute nextVersionNo is a counter which stores the next available version number at the versioned object. The second rule creates a successor version, which is assigned the next available version number, and increments the counter. Using only the first and the second rule, only composition trees may be created in the repository. In the third rule, a directory is created in the workspace and related to a reused version which is added to the version of the parent directory.

Unfortunately, the third rule (CreateDirectoryAndReuseVersion) does not operate correctly. After its execution, the composition hierarchy rooted at the new directory in the workspace is empty, but this does not necessarily apply to the reused version at the other end of the link. If the composition hierarchy below the reused version is not empty, the generation process will "get of out sync". This problem could be fixed by adding directed rules which copy the composition hierarchy into the workspace, but this would break the TGG paradigm (relying on synchronous rules only).

The style of modeling employed in the TGG is quite different from the style of the TGTS. The TGG consists of productions which are partially obtained by copying productions from the grammars for the source and target graphs, respectively (*white-box reuse*). In contrast, in the TGTS source and target graph may be read, but they may be updated only through method calls. This *grey-box reuse* avoids duplication of consistency checks and transformations.

Directed Rules. While synchronization of repositories and workspaces involves bidirectional change propagation with forward and backward rules, *correspondence rules* are of little use: An analysis tool which discovers correspondences between repositories and workspaces and extends the correspondence graph accordingly is not required. The status check, which analyzes consistency between repository and workspace, merely determines the status of already existing link objects (see next paragraph).

Forward rules are obtained from synchronous triple rules by converting created elements in the source graph into elements of the left-hand side. For the rules of Figure 6, this means that the directory nd and its composition link have to be moved to the left-hand side (i.e., they must already be present to apply the rule).

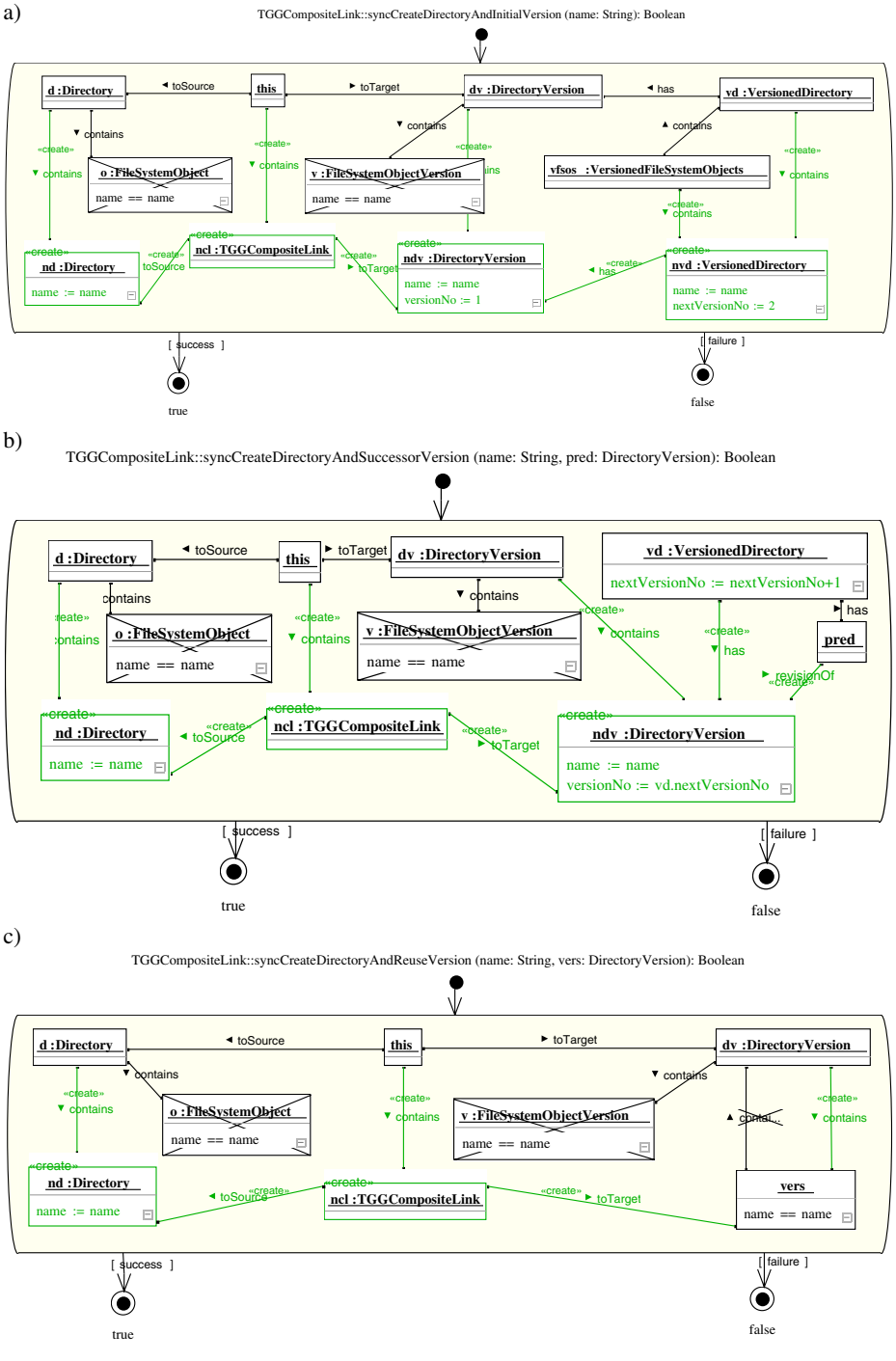


Fig. 6. Synchronous rules for creating directories and directory versions

Since all rules shown in Figure 6 are identical with respect to the source graph and the correspondence graph, the generated rules stand in *conflict*: A new subdirectory can be transformed by any of these rules, and applying one rule invalidates the other choices. Since the rules have different effects, the generated TGTS is *non-deterministic*, resulting in an integration tool which requires user interaction. In contrast, the commands introduced in Section 2 operate in a *deterministic* way. For a new file system object, the user may not deliberately choose either to create a new versioned object and its root version, or to create a successor version, or to reuse an already existing version. Rather, only the first option is available for a new file system object. A successor version is created when the file system object has already been linked to a version in the repository and has been changed locally in the workspace. Finally, a version is reused when a new version of the parent directory has been created, the child object is already under version control and has not been changed in the workspace. Similar problems occur with respect to backward rules; these cannot be elaborated here due to space restrictions.

Another problem which we encountered in our case study consists in the assumption of the TGG approach that forward and backward transformations operate *symmetrically*: From a single synchronous rule, symmetric forward and backward rules are derived. However, forward and backward transformations behave differently in our case study. For example, when the content of a file is modified in the workspace, a new version is created in the repository. In contrast, when the new version is propagated into another workspace by running an update command, the file in the workspace is overridden. Furthermore, composition hierarchies are treated differently in the workspace (tree) and in the repository (acyclic graph). As a consequence, an acyclic graph is *unfolded* into a tree when it is exported into a workspace, and a tree in the workspace is *folded* into an acyclic graph when changes are committed into the repository.

Consistency Checks and Repair Actions. One of the most important goals of the TGG approach is to relieve the modeler from the burden of specifying modifications and deletions. Since the TGG rules define only graph extensions, this abstraction works only when all operations concerning modifications and deletions can be derived automatically. To this end, generic support for *consistency checks* and *repair actions* is required (see Figure 1). Unfortunately, to the best of our knowledge providing such checks and repair actions in a generic way still constitutes an open research problem.

In our case study, consistency checks are performed in the checkStatus command. Some parts of the status checks could be covered by a generic approach, e.g., differences between values of attributes such as file names and file contents. However, there are some parts which are highly domain-specific. For example, changes in some file system object need to be propagated bottom-up to the root such that new versions in the repository may be created top-down in the course of a commit. Furthermore, the status check has to recognize updates in the repository that have to be propagated into the workspace. This is a domain-specific requirement, and there is no triple rule from which we could derive such a check. Repair actions are performed in the commands for synchronization, namely update and commit. Again, these repair actions are domain-specific. For example, when a file in the workspace is deleted, its corresponding version in the repository is not deleted, as well. Rather, a new version of the parent directory is created which does not contain this file version.

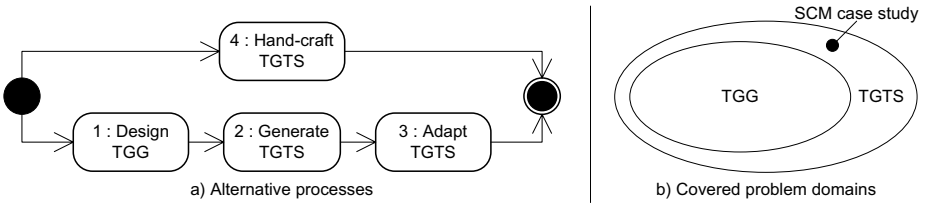


Fig. 7. Triple graph grammars or triple graph transformation systems?

5 Related Work

The overall goal of the MOD2-SCM project is to provide a model-driven product line that allows to construct a wide range of customized SCM systems with acceptable effort. These goals are related to a few research projects which were dedicated to the development of a *uniform version model*. Both ICE [9] and UVM [10] proposed rule-based generalizations of conditional compilation as a low-level, common base mechanism. To date, only a few approaches have been dedicated to *model-driven development of versioning systems* [11,12]. However, these approaches are confined to structural models inasmuch as the behavior is hard-coded into the respective system.

Triple graph grammars were introduced as early as 1994 [4]. The QVT standard [1], which was developed much later, is based on similar concepts. In the context of this paper, it is interesting to note that QVT defines both a high-level declarative and a more low-level operational language. Several projects were built upon the concepts of TGGs, but actually developed a hand-crafted TGTS [13,14]. Frameworks supporting code generation for TGGs have emerged only recently [15,16,17]. Surveys of the current state-of-the-art in TGGs are given in [5,18]. In [19], research challenges and open problems are discussed. Four design principles for TGGs are postulated: completeness, consistency, efficiency, and expressiveness. The case study presented in this paper primarily challenges the expressiveness of TGGs (see the conceptual problems reported in Section 4).

6 Conclusion

For the synchronization between repositories and workspaces, we discussed the alternatives of hand-crafting a TGTS or generating it from a TGG (Figure 7a). Following the TGG process of Figure 1, the costs of step 2 would be zero (automatic step) and step 3 would be obsolete. However, this process did not work in our case study; it was more effective to hand-craft the TGTS (step 4) than to define a TGG and adapt the generated TGTS. In fact, the case study belongs to the range of problems which are not suited for applying the TGG approach (Figure 7b). For various reasons, the operational behavior of generated rules does not match the requirements of our case study. Thus, the region $TGTS \setminus TGG$ is not empty. Improvements from theory may reduce this region further, but more practical case studies are also needed to explore the potentials and limitations of the TGG approach.

References

1. Object Management Group Needham, Massachusetts: Meta Object Facility (MOF) 2.0 Query/View/Transformation Specification. Final adopted specification ptc/07-07-07 edn. (July 2007)
2. Rozenberg, G. (ed.): Handbook on Graph Grammars and Computing by Graph Transformation: Foundations, vol. 1. World Scientific, Singapore (1997)
3. Ehrig, H., Engels, G., Kreowski, H.J., Rozenberg, G. (eds.): Handbook on Graph Grammars and Computing by Graph Transformation: Applications, Languages, and Tools, vol. 2. World Scientific, Singapore (1999)
4. Schürr, A.: Specification of graph translators with triple graph grammars. In: Mayr, E.W., Schmidt, G., Tinhofer, G. (eds.) WG 1994. LNCS, vol. 903, pp. 361–375. Springer, Heidelberg (1995)
5. Königs, A., Schürr, A.: Tool Integration with Triple Graph Grammars - A Survey. Electronic Notes in Theoretical Computer Science 148, 113–150 (2006)
6. Buchmann, T., Dotor, A., Westfechtel, B.: MOD2-SCM: Experiences with co-evolving models when designing a modular SCM system. In: Proceedings of the 1st International Workshop on Model Co-Evolution and Consistency Management, Toulouse, France (2008)
7. Zündorf, A.: Rigorous object oriented software development. Technical report, University of Paderborn, Germany (2001)
8. Conradi, R., Westfechtel, B.: Version models for software configuration management. ACM Computing Surveys 30(2), 232–282 (1998)
9. Zeller, A., Snelting, G.: Unified versioning through feature logic. ACM Transactions on Software Engineering and Methodology 6(4), 397–440 (1997)
10. Westfechtel, B., Munch, B.P., Conradi, R.: A layered architecture for uniform version management. IEEE Transactions on Software Engineering 27(12), 1111–1133 (2001)
11. Whitehead, E.J., Ge, G., Pan, K.: Automatic generation of hypertext system repositories: a model driven approach. In: 15th ACM Conference on Hypertext and Hypermedia, pp. 205–214. ACM Press, New York (2004)
12. Kovše, J.: Model-Driven Development of Versioning Systems. PhD thesis, University of Kaiserslautern, Kaiserslautern, Germany (August 2005)
13. Jahnke, J., Zündorf, A.: Applying graph transformations to database re-engineering. In: [3], pp. 267–286
14. Cremer, K., Marburger, A., Westfechtel, B.: Graph-based tools for re-engineering. Journal of Software Maintenance and Evolution: Research and Practice 14(4), 257–292 (2002)
15. Becker, S.M., Herold, S., Lohmann, S., Westfechtel, B.: A graph-based algorithm for consistency maintenance in incremental and interactive integration tools. Software and Systems Modeling 6(3), 287–315 (2007)
16. Giese, H., Wagner, R.: Incremental model synchronization with triple graph grammars. In: Nierstrasz, O., Whittle, J., Harel, D., Reggio, G. (eds.) MoDELS 2006. LNCS, vol. 4199, pp. 543–557. Springer, Heidelberg (2006)
17. Amelunxen, C., Klar, F., Königs, A., Rötschke, T., Schürr, A.: Metamodel-based tool integration with MOFLON. In: 30th International Conference on Software Engineering, pp. 807–810. ACM Press, New York (2008)
18. Kindler, E., Wagner, R.: Triple graph grammars: Concepts, extensions, implementations, and application scenarios. Technical Report tr-ri-07-284, University of Paderborn, Paderborn, Germany (June 2007)
19. Schürr, A., Klar, F.: 15 Years of Triple Graph Grammars — Research Challenges, New Contributions, Open Problems. In: Ehrig, H., Heckel, R., Rozenberg, G., Taentzer, G. (eds.) ICGT 2008. LNCS, vol. 5214, pp. 411–425. Springer, Heidelberg (2008)