The Added Value of Programmed Graph Transformations – A Case Study from Software Configuration Management

Thomas Buchmann, Bernhard Westfechtel, and Sabine Winetzhammer

Lehrstuhl Angewandte Informatik 1, University of Bayreuth
D-95440 Bayreuth, Germany
firstname.lastname@uni-bayreuth.de

Abstract. Model-driven software engineering intends to increase the productivity of software engineers by replacing conventional programming with the development of executable models at a high level of abstraction. It is claimed that graph transformation rules contribute towards this goal since they provide a declarative, usually graphical specification of complex model transformations. Frequently, graph transformation rules are organized into even more complex model transformations with the help of control structures, resulting in full-fledged support for executable behavioral models.

This paper examines the added value of programmed graph transformations with the help of a case study from software configuration management. To this end, a large model is analyzed which was developed in the MOD2-SCM project over a period of several years. The model was developed in Fujaba, which provides story diagrams for programming with graph transformations. Our analysis shows that the model exhibits a strongly procedural flavor. Graph transformation rules are heavily used, but typically consist of very small patterns. Furthermore, story diagrams provide fairly low level control structures. Altogether, these findings challenge the claim that programming with graph transformations is performed at a significantly higher level of abstraction than conventional programming.

1 Introduction

Model-driven software engineering is a discipline which receives increasing attention in both research and practice. Object-oriented modeling is centered around class diagrams, which constitute the core model for the structure of a software system. From class diagrams, parts of the application code may be generated, including method bodies for elementary operations such as creation/deletion of objects and links, and modifications of attribute values. However, for user-defined operations only methods with empty bodies may be generated which have to be filled in by the programmer. Here, programmed graph transformations may provide added value for the modeler. A behavioral model for a user-defined operation may be specified by a programmed graph transformation. A model instance being composed of objects, links, and attributes is considered as an attributed graph. A graph transformation rule specifies the replacement of a subgraph in a declarative way. Since it may not be possible to model the behavior of a user-defined

A. Schürr, D. Varró, and G. Varró (Eds.): AGTIVE 2011, LNCS 7233, pp. 198–209, 2012.

operation with a single graph transformation rule, control structures are added to model composite graph transformations.

But what is the added value of programmed graph transformations? Typical arguments which have been frequently repeated in the literature in similar ways are the following ones:

1. A graph transformation rule specifies a complex transformation in a rule-based, declarative way at a much higher level of abstraction than a conventional program composing elementary graph operations.
2. Due to the graphical notation, programming with graph transformations is intuitive and results in (high-level) programs which are easy to read and understand.

This paper examines the added value of programmed graph transformations by analyzing a large model which was developed in the MOD2-SCM project (Model-Driven and Modular Software Configuration Management) [3] over a period of several years. The model was developed in Fujaba [8], which provides story diagrams for programming with graph transformations. We analyze the MOD2-SCM model both quantitatively and qualitatively to check the claims stated above.

2 MOD2-SCM

The MOD2-SCM project [3] is dedicated to the development of a model-driven product line for Software Configuration Management (SCM) systems [1]. In contrast to common SCM systems, which have their underlying models hard-wired in hand-written program code, MOD2-SCM has been designed as a modular and model-driven approach which (a) reduces the development effort by replacing coding with the creation of executable models and (b) provides a product line supporting the configuration of an SCM system from loosely coupled and reusable components.

To achieve this goal, we used Fujaba [8] to create the executable domain model of the MOD2-SCM system. The main part of the work was to (1) create a feature model [4], that captures the commonalities and variable parts within the domain software configuration management and (2) to create a highly modular domain model whose loosely coupled components can be configured to derive new products. To this end, a model library consisting of loosely coupled components that can be combined in an orthogonal way has been built around a common core.

The success of a product line heavily depends upon the fact that features that have been declared as independent from each other in the feature model are actually independent in their realizing parts of the domain model. Thus, a thorough analysis of the dependencies among the different modules is crucial [3] in order to derive valid product configurations. In order to keep track of the dependencies in large domain models, a tool based upon UML package diagrams has been developed and integrated with Fujaba to support the modeler during this tedious task [2]. In the context of the MOD2-SCM project, graph transformations were used to specify the behavior of the methods that have been declared in the domain model.

In this paper, we will discuss the added value of graph transformations especially in the development of large and highly modular software systems. The added value

of Fujaba compared with other CASE tools is the ability to generate executable code out of behavioral models. Behavioral modeling in Fujaba is performed with story diagrams which are similar to interaction overview diagrams in UML2 [6]. Within story diagrams, activities and transitions are used to model control flow. Fujaba supports two kinds of activities: (1) statement activities, allowing the modeler to use source code fragments that are merged 1:1 with the generated code, and (2) story activities. A story activity contains a story pattern consisting of a graph of objects and links. A static pattern encodes a graph query, a pattern containing dynamic elements represents a graph transformation rule. Story patterns may be decorated with constraints and collaboration calls. A story activity may be marked as a "for each" activity, implying that the following activities are performed for each match of the pattern. In addition to activity nodes, story diagrams contain start, end, and decision nodes.

In the following section, we analyze the domain model of MOD2-SCM with a specific focus on story diagrams.

3 Analysis

3.1 Quantitative Analysis

Tool support was required to analyze the structure and complexity of the MOD2-SCM specification. Due to the size of the project, determining the numbers listed in Tables 1 and 2 would have been a tedious task. Therefore, we wrote a Fujaba plug-in that directly operates on Fujaba's abstract syntax graph to acquire the numbers we were interested in. We conclude from the collected numbers:

Table 1. Type and number of language elements

Model element (structural model)	Total number MOD2-SCM	Total number CodeGen2	Model element (behavioral model)	Total number MOD2-SCM	Total number CodeGen2
Packages	68	18	*Story diagrams*	540	339
Classes	175	162	*Story patterns*	988	850
Abstract Classes	18	28	*Objects*	1688	1997
Interfaces	32	10	*Negative objects*	42	22
Attributes	177	181	*Multi-objects*	25	9
Methods	650	443	*Links*	725	1121
Generalizations	220	247	*Paths*	13	7
Associations	148	166	*Statement activities*	264	64
			Collaboration calls	1183	711
			For each activities	27	88

1. The number of story patterns per story diagram is rather low (an average of 1.83 story patterns were used per story diagram). This indicates a procedural style of the model and methods of lower complexity.

Table 2. Significant ratios of language elements

Metric	MOD2-SCM	CodeGen2	Metric	MOD2-SCM	CodeGen2
Classes/package	2.57	9	*Negative objects/pattern*	0.04	0.03
Attributes/class	1.01	1.12	*Multi-objects/pattern*	0.03	0.01
Methods/class	3.71	2.73	*Paths/pattern*	0.01	0.01
Patterns/story diagram	1.83	2.51	*Statement activities/story pattern*	0.27	0.08
Objects/pattern	1.71	2.35	*Collaboration calls/story pattern*	1.20	0.84
Links/pattern	0.73	1.32			

2. Within a story pattern, only a few objects and links are used (1.71 objects and 0.73 links, respectively).
3. The number of collaboration calls is rather high (an average number of 1.2 collaboration calls per pattern was determined).
4. Given the fact that we tried to use story patterns as much as possible, the fraction of statement activities is still rather high (0.27 statement activities per story activity). Furthermore, the seamless integration of hand-written Java code and story patterns provides lots of advantages, but it also implies one big disadvantage: The model is no longer platform-independent. For example, changing the code generation templates to generate C# code is no longer possible without manually changing each statement activity in the domain model as well.
5. Complex elements within story patterns (negative objects, multi-objects or paths) were used only very rarely within the domain model.

Generally, the specification of the MOD2-SCM system is highly procedural and essentially lies at the same level of abstraction as a conventional program. The average complexity of the implemented methods is rather low. Furthermore, the graph transformation rules are mostly limited to finding simple patterns in the object graph and to inserting a single new object at the appropriate position and/or calling another method.

3.2 Qualitative Analysis

In addition to interpreting the numbers shown in the previous section, we took a closer look at the story diagrams of the domain model to examine the expressive power of the modeling language and the readability of story diagrams. With respect to story patterns, we observed:

6. Story patterns are easy to read due to the graphical notation. Furthermore, story patterns potentially have a high expressive power, but this power is only rarely exploited since the highly modular architecture results in a fine-grained decomposition of the domain model.

We drew the following conclusions concerning control flow:

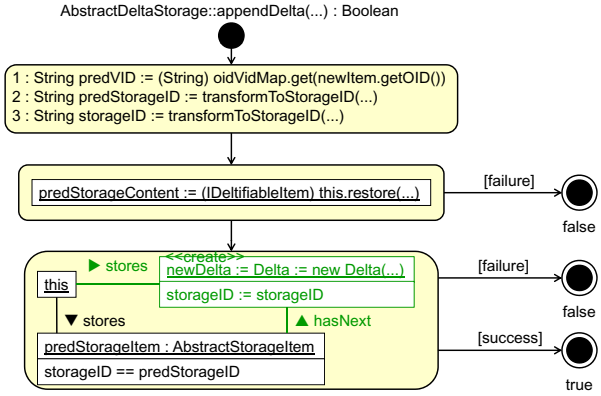

Fig. 1. Story diagram of average complexity

7. It is evident that Fujaba does not provide any "higher"-level control structures. In fact, in terms of control structures, the level of abstraction provided by Fujaba lies even below conventional programming languages, as story diagrams are very similar to flow charts.
8. In many cases, Java statement activities are mixed up with story patterns, e.g., for exception handling. Fujaba does not catch external exceptions raised in the execution of story patterns, e.g., by external Java methods executed via collaboration calls. Such low-level details need to be handled in Java.
9. Furthermore, Fujaba itself does not provide any mechanisms to iterate over standard collections. Collections are implicitly used within for-each activities, but no explicit support is provided for the user.
10. The graphical programming style may result in loss of the overall picture if the diagrams are too large and complex. Readability suffers especially (but not only) when the story diagram does not fit onto a single screen.

4 Examples

In this section, we give some examples that reinforce the statements of the previous section. Figure 1 represents a method implementation which we consider to be of *average complexity* (according to the collected metrics data). It is a typical example for the observations (1) – (5) made in Section 3.1 as well as observation (6). The method is used to append a forward delta in order to store a new state of a versioned object. The first story pattern consists only of collaboration calls that retrieve different required parameters. The second pattern retrieves the content of the predecessor version (which must have been stored as a delta). This content is used in the third pattern (the only graph transformation rule) to compute the difference and to store it at the appropriate position in the object graph.

All story patterns occurring in Figure 1 are quite simple. Figure 2 shows a *(moderately) complex story pattern* which stores a backward delta. The enclosing story diagram is called when a version stored as baseline is to be replaced with a successor

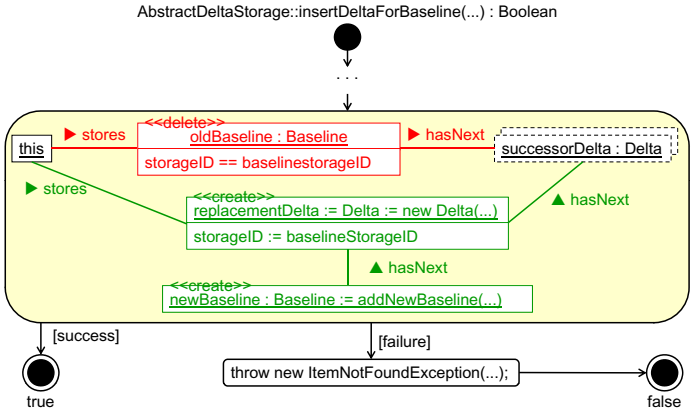

Fig. 2. A (moderately) complex story pattern

version. To this end, the old baseline is replaced with a new baseline and a backward delta. In fact this is one of the most complex patterns throughout the whole MOD2-SCM model, which does not include any patterns with more than 6 objects. This example illustrates a strength of story patterns: The effect of a graph transformation is documented in an easily comprehensible way. However, most patterns are much simpler, implying that the equivalent Java code would as well be easy to grasp (observation (6)).

Figure 3 shows the story diagram with the *highest number of story patterns* throughout the MOD2-SCM project. The story diagram is used to update the local version information within the user's workspace after changes have been committed to the server. This method is very complex if we take the average number of patterns per story diagram into account. However, its individual steps are not complex at all. Listing 1 shows the purely hand-written implementation in Java. Essentially, the method consists of a single loop iterating over a list of object identifiers. Two of the story patterns contain plain Java code since Fujaba does not supply high level constructs for iterating over standard collections (observation (9)). The remaining story patterns are also very simple (observation (5)). Thus, it is not a big challenge to code this story diagram in Java.

The story diagrams presented so far do not contain statement activities, except for a small activity in Figure 2. We used statement activities only when they were impossible or awkward to avoid. Observation (4) showed that our attempt to eliminate story patterns was only successful to a limited extent. The next two examples demonstrate the reasons for that.

The first example (Figure 4) shows a story diagram which is used to configure the MOD2-SCM repository server at runtime according to the features selected by the user. This method is *inherently procedural* and consists of a large number of conditional statements for handling the different cases. The modeler decided to code the method body as a single statement activity. Splitting this activity up into many decision nodes and activity nodes containing small code fragments would have resulted in a huge and unreadable diagram (observation (7)).

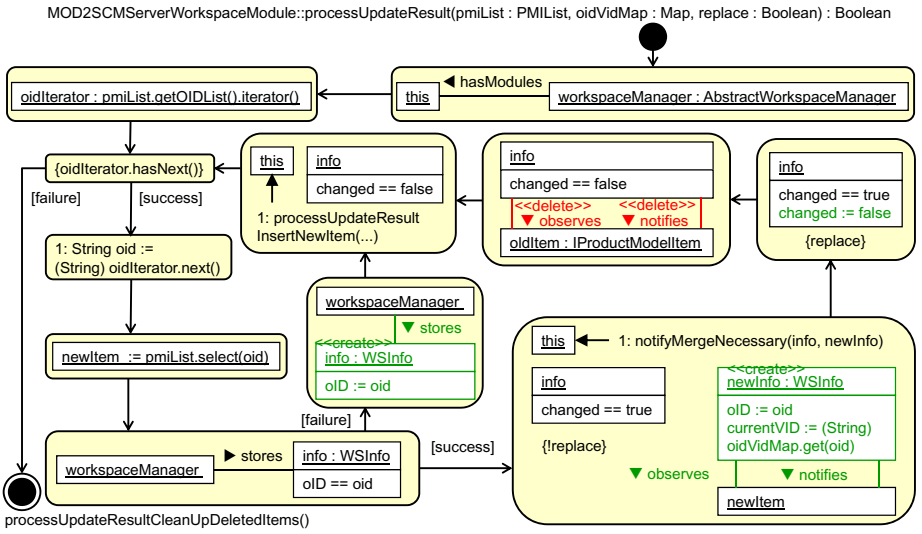

Fig. 3. Story diagram that contains a high number of patterns

Listing 1. Manual implementation of the method shown in Figure 3

```
1  public boolean processUpdateResult(PMIList pmiList, Map<String, String>
       oidVidMap, boolean replace) {
2    Iterator oidIterator = pmiList.getOIDList().iterator();
3    while (oidIterator.hasNext()) {
4      String oid = (String) oidIterator.next();
5      IProductModelItem newItem = pmiList.select(oid);
6      WSInfo info = getWorkspaceManager().getWSInfos().get(oid);
7      if (info != null) {
8        IProductModelItem oldItem = info.getItem();
9        if (!replace && info.isChanged()) {
10         WSInfo newInfo = new WSInfo(oid, (String) oidVidMap.get(oid));
11         newInfo.setItem(newItem);
12         newInfo.setObservable(newItem);
13         notifyMergeNecessary(info, newInfo);
14       }
15       if (replace && info.isChanged())
16         info.setChanged(false);
17       if (!info.isChanged() && oldItem != null) {
18         info.setItem(null);
19         info.setObservable(null);
20       }
21     } else {
22       info = new WSInfo(oid);
23       getWorkspaceManager().addWSInfo(info);
24     }
25     if (!info.isChanged())
26       processUpdateResultInsertNewItem(oldItem, newItem, info, oidVidMap);
27   }
28   processUpdateResultCleanUpDeletedItems();
29 }
```

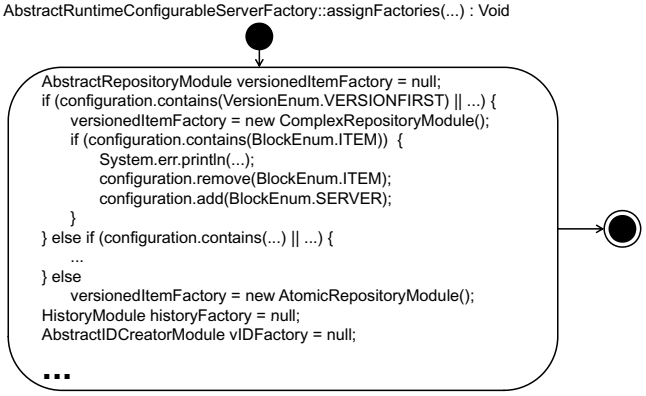

Fig. 4. Story diagram that represents a highly procedural example

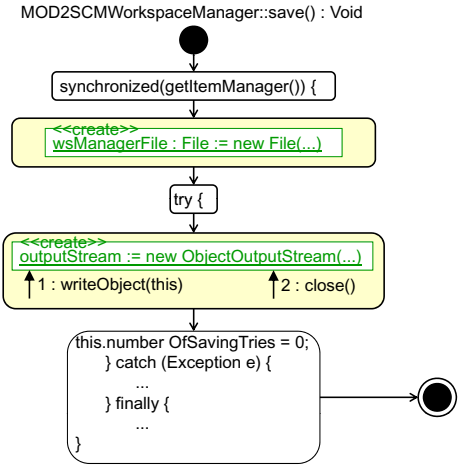

Fig. 5. Story diagram that illustrates the JSP syndrome

Figure 5 is a good example how hand-written code fragments are placed around story patterns (observation (8)). The story diagram is used to save the state of the workspace manager. To ensure that the execution is synchronized, the body is embedded into a synchronization statement. Furthermore, if the write operation fails, the save method is re-executed (until the maximum number of tries is exceeded). Again, this method implementation is highly procedural and story driven modeling does not seem to be the most appropriate formalism for this task. The story diagram is written in a *JSP-like style*, including statement activities containing fragments of Java text which do not even correspond to complete syntactical units.

Figure 6 shows a story diagram that is used to calculate differences on text files based upon the well-known Longest Common Subsequence (LCS) algorithm. Figure 6 depicts

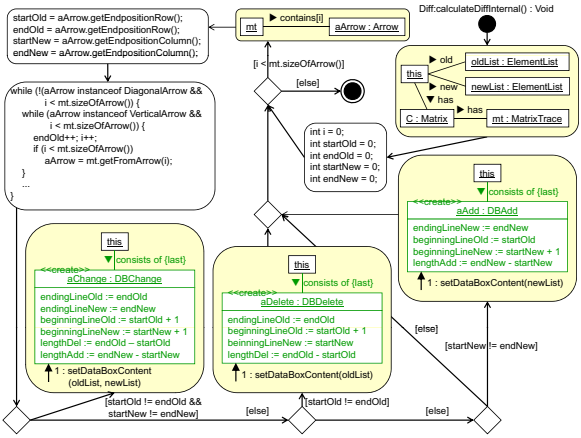

Fig. 6. Story diagram for the LCS algorithm which may be coded easily in Java

the part of the algorithm where the LCS has been determined and the difference (represented by add, change, or delete blocks) is being computed. The original algorithm performs best when working with arrays that contain the indices of the longest common subsequences in two text fragments. Nevertheless, the student who implemented this part of the MOD2-SCM system tried to raise the algorithm to the object-oriented level and to make use of Fujaba to be able to easily integrate it into the already existing MOD2-SCM domain model. Working with indices was still necessary, though objects for the matrices and the trace within the matrix representing the longest common subsequence have been created. The result is a mixture of statement activities and story patterns, which makes it hard to keep track of the actual control flow within the shown story diagram (*unstructured control flow*, observation (10)).

5 Discussion

5.1 Results from the MOD2-SCM Project

In this paper, we examined the added value of programmed graph transformations with the help of a case study from software configuration management. Based on this case study, may we convince hard-core Java programmers to program with graph transformations instead? The examination of the story diagrams developed in the MOD2-SCM project suggests the answer “no”. Altogether, story diagrams are written in a strongly procedural style at a level of abstraction which hardly goes beyond Java and is even partially located below Java or other current programming languages.

The authors of the MOD2-SCM model made extensive use of story patterns. However, our quantitative analysis showed that story patterns are typically composed of a very small number of objects and links. Furthermore, advanced features such as negative objects/links, multi-objects, and paths are only very rarely used. Altogether, the potential of story patterns - the declarative specification of complex graph transformations - is only exploited to a severely limited extent.

As far as story patterns are concerned, the graphical notation is intuitive and enhances readability, in particular in the case of more complex graph transformations. With respect to control flow, however, the graphical notation may have a negative impact on readability. Essentially, story diagrams are conventional flow charts, which are well known for the "goto considered harmful" syndrome. Control structures known from structured programming are missing. In this respect, story diagrams fall behind conventional programming languages such as Java.

5.2 Generalization of Results

Let us summarize the most important general observations derived from the case study:

1. The behavioral model is highly procedural.
2. The expressive power of graph transformation rules (story patterns) is hardly exploited.

It might be argued that these findings are specific to the case study since providing a product line requires the fine-grained decomposition of the overall domain model into a set of rather small reusable components. However, this style of development is not only applied to product lines, but it should anyhow be applied in any large development project. To check this assumption, we ran our metrics tool on Fujaba's CodeGen2 model (the bootstrapped Fujaba code generator). The results were very similar to the data collected from the MOD2-SCM project (see again Tables 1 and 2).

It could also be argued that the authors of the story diagrams lacked the expertise to fully exploit the features of the modeling language. Although some minor parts of the MOD2-SCM project were developed by students who did not have much experience in programming with graph transformations, the biggest part of the system was implemented by experienced Fujaba modelers. Furthermore, the analysis of CodeGen2, which was developed by the authors of Fujaba themselves, yielded similar results.

Finally, it might be argued that the procedural style of the Fujaba models is due to the modeling language. However, this argument does not explain the fact that advanced features of story patterns such as paths, constraints, negative objects and links, and multi-objects were only rarely used. Nevertheless, we decided to examine a large specification written in another language to check this argument. The specification was developed in a Ph.D. thesis in the ECARES project, which was concerned with reverse engineering and reengineering of telecommunication systems [5]. The specification was written in PROGRES [7], a language for programming with graph transformations providing many advanced features (multiple inheritance, derived attributes and relations, object-orientation, genericity, graph transformation rules with similar features as in Fujaba, high-level control structures, backtracking, and atomic transactions).

The data displayed in Tables 3 and 4 were collected from the complete specification, as developed in the Ph.D. thesis by Marburger. By and large, the results are consistent with the metrics data collected from the Fujaba models:

1. The ECARES specification has a strongly procedural flavor. This is indicated by the ratio of the number of graph tests and graph transformation rules related to the number of programmed graph queries and transactions: There are twice as many programmed methods as elementary graph tests and transformation rules.

Table 3. Type and number of language elements in ECARES

Model element	Total number	Model element	Total number
Packages	21	*Optional nodes*	9
Node classes or types	190	*Set nodes*	40
Generalizations	193	*Edges*	374
Intrinsic attributes	87	*Negative nodes and edges*	13
Derived attributes	9	*Positive and negative paths*	92
Meta attributes	8	*Transactions (update methods)*	299
Edge types	21	*Queries*	8
Textual path declarations	32	*Assignments*	659
Graphical path declarations	40	*Calls*	684
Graph tests	55	*Sequences*	335
Graph transformation rules	92	*Conditional statements*	270
Mandatory nodes	559	*Loops*	83

Table 4. Significant ratios of language elements in ECARES

Metric	Value	Metric	Value
(Classes + types)/package	9.05	*(Negative nodes + edges)/graphical definitions*	0.07
Attributes/(classes + types)	0.55	*Set nodes/graphical definitions*	0.21
(Graph tests + graph transformation rules + queries + transactions)/(classes + types)	2.39	*(Positive + negative paths)/graphical definitions*	0.49
(Graph tests + graph transformation rules)/ (queries + transactions)	0.48	*(Assignments + calls)/(queries + transactions)*	4.37
Nodes/graphical definitions	2.99	*Control structures/(queries + transactions)*	2.24
Edges/graphical definitions	2.00		

2. Graphical definitions (graph tests, graph transformation rules, and graphical path declarations) are rather small. On average, a graphical definition contains about 3 nodes and 2 edges. These numbers are a bit larger than in MOD2-SCM and CodeGen2. However, it has to be taken into account that relationships are represented in ECARES always as nodes and adjacent edges. Thus, a graph transformation rule for inserting a relationship requires at least 3 nodes and 2 edges. In the publications on ECARES, considerably more complex rules were selected for presentation, but these rules are not representative.
3. The data differ with respect to the utilization of advanced features in graphical definitions. In particular, paths are used in about 50% of all graphical definitions. Eliminating paths would result in larger graphical definitions. Thus, altogether the graphical definitions are slightly more complex than in the studied Fujaba models.

6 Conclusion

We investigated the added value of programmed graph transformations with the help of a large case study from software configuration management. Our analysis showed that the model developed in the MOD2-SCM project exhibits a strongly procedural flavor. Furthermore, graph transformation rules are heavily used, but consist typically of very

small and simple patterns. Finally, we have reinforced our findings with metrics data collected from other projects utilizing programmed graph transformations.

Examining a few large models does not suffice to evaluate the added value of programmed graph transformations. However, our analysis indicates that the level of abstraction is not raised as significantly as expected in comparison to conventional programming. In the models we studied, the modeling problem seems to demand for a procedural solution. Furthermore, modularization of a large model may result in a fine-grained decomposition such that each method only has to deal with small patterns and has to provide a small piece of functionality. Further case studies are required to check whether these effects also apply to other applications.

References

1. Buchmann, T., Dotor, A.: Towards a model-driven product line for SCM systems. In: McGregor, J.D., Muthig, D. (eds.) Proc. of the 13th Int. Software Product Line Conference, vol. 2, pp. 174–181. SEI (2009)
2. Buchmann, T., Dotor, A., Klinke, M.: Supporting modeling in the large in Fujaba. In: van Gorp, P. (ed.) Proc. of the 7th International Fujaba Days, pp. 59–63 (2009)
3. Dotor, A.: Entwurf und Modellierung einer Produktlinie von Software-Konfigurations-Management-Systemen. Ph.D. thesis, University of Bayreuth (2011)
4. Kang, K.C., Cohen, S.G., Hess, J.A., Novak, W.E., Peterson, A.S.: Feature-oriented domain analysis (FODA) feasibility study. Tech. rep., Carnegie-Mellon University Software Engineering Institute (1990)
5. Marburger, A.: Reverse Engineering of Complex Legacy Telecommunication Systems. Berichte aus der Informatik. Shaker-Verlag (2005)
6. OMG: OMG Unified Modeling Language (OMG UML), Superstructure V2.2 (version 2.2) (February 2009)
7. Schürr, A., Winter, A., Zündorf, A.: The PROGRES Approach: Language and Environment. In: Handbook of Graph Grammars and Computing by Graph Transformation, vol. 2: Applications, Languages, and Tools, pp. 487–550. World Scientific (1999)
8. Zündorf, A.: Rigorous object oriented software development. Tech. rep., University of Paderborn, Germany (2001)