Structuring Complex Story Diagrams by Polymorphic Calls

Technical Report tr-ri-11-323

Steffen Becker, Markus von Detten, Christian Heinzemann*, and Jan Rieke*

Software Engineering Group, Heinz Nixdorf Institute
University of Paderborn, Paderborn, Germany
[steffen.becker|mvdetten|c.heinzemann|jrieke]@uni-paderborn.de

Abstract. In model-driven software engineering, model transformations occur frequently, e.g., to transform platform independent models into platform specific models. Hybrid languages using imperative and declarative elements seem to be a promising approach as they integrate explicit control-flow with efficient matching. Story diagrams are such a hybrid model transformation language that combines UML activity diagrams and graph transformations in a graphical transformation language. Hitherto, story diagrams do not support structuring complex transformations into several independent story diagrams which can be called in a well-defined manner. This prevents rule reuse and hence significantly reduces the maintainability of the transformations. In this paper, we therefore extend story diagrams by explicit call activities that invoke other story diagrams and support polymorphic dispatching. We evaluate the benefits of such calls by revisiting an already implemented model transformation and discussing improvements due to our new call concept.

Keywords: Visual Model Transformation, Graph Transformation, Control Flow, Polymorphic Dispatching

1 Introduction

In model-driven software engineering, model transformations play a central role in transforming models of higher abstraction levels into more concrete models. Such transformations are written in special purpose languages which offer explicit support for common transformation tasks like matching elements of the source model. While the development of current model transformation languages focused on those model transformation specific tasks, classical issues like inheritance and structuring were neglected. As a result, the transformations are often concise and efficient but hardly maintainable.

Story diagrams [5] form a special model transformation language. They feature declarative parts to specify object patterns which are matched and altered in

* supported by the International Graduate School Dynamic Intelligent Systems.

the source model and combine them with an imperative part to specify the control flow of the transformation execution. The concrete syntax of story diagrams extends the concrete syntax of UML activity diagrams. A specific challenge for story diagrams is the missing support for the invocation of other story diagrams. We require these invocations to account for aspects like the increased complexity of binding parameters and result values in a graphical language as well as dynamic dispatching of calls.

Some of the related transformation languages like ATL, QVT Operational, or Henshin already offer some support for structuring transformations into sub transformations. However, no existing hybrid and graphical transformation languages addresses all of the aforementioned challenges.

In this paper, we present an approach to tackle the lacking feature of invoking story diagrams. Our solution supports binding the parameter values as well as the result values of the invoked diagram. We also introduce polymorphic dispatching in our solution, i.e., the invoked diagram is chosen at run-time based on the actual types of the bound parameter values. Consequently, the story diagram meta-model as well as the existing interpreter for story diagrams [8] have to be extended to support the new concepts.

We show the effectiveness of our approach in a qualitative study of an existing transformation implemented in story diagrams. In the study, we estimate the potential to reduce the total amount and size of the story diagrams in our initial implementation. The results show a significant improvement in transformation size, leading to reduced complexity and thus an increase in the long-term maintainability of the transformation.

The contribution of this paper is an approach to include invocations of other story diagrams. We present a running example, necessary extensions of the story diagram meta-model and a case study to show the effectiveness of our approach.

The paper is structured as follows. The following section presents an illustrative running example used throughout the paper to explain the new story diagram concepts. Section 3 introduces the foundations of the story diagram language. In Section 4, we extend these concepts and introduce calls to other story diagrams. Section 5 presents an evaluation of the new concepts in the context of a real transformation. After discussing related approaches in other languages in Section 6, we conclude our paper and highlight open issues.

2 Illustrative Example

To illustrate our concept, we use a language transformation scenario: Suppose we have a model representing the abstract syntax graph (ASG) of a program. The ASG is language-specific, of course. If we want to generate code from the ASG for a target language other than the language on which the given ASG is based, one option is to transform the ASG first. One example of this would be a transformation from Java to C++.

Figure 1 shows a simplified meta-model for an ASG which acts as the source meta-model for our exemplary transformation scenario. A SourceSystem consists

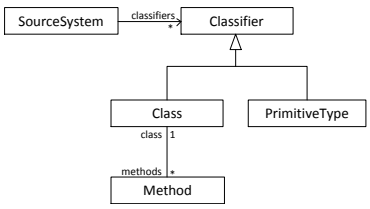

Fig. 1. Source meta-model

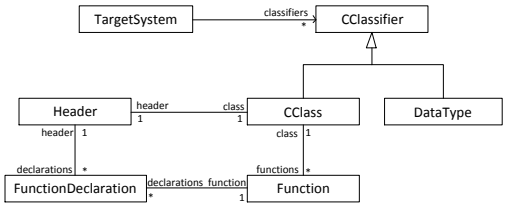

Fig. 2. Target meta-model

of a number of Classifiers which can either be PrimitiveTypes or Classes. Each Class can have a number of Methods.

The target meta-model for the transformation, shown in Figure 2, is slightly more complex. The TargetSystem consists of a number of CClassifiers which are either DataTypes or CClasses. An CClass can contain a number of Functions. In addition, each CClass has a corresponding Header which contains a FunctionDeclaration for each Function of the CClass.

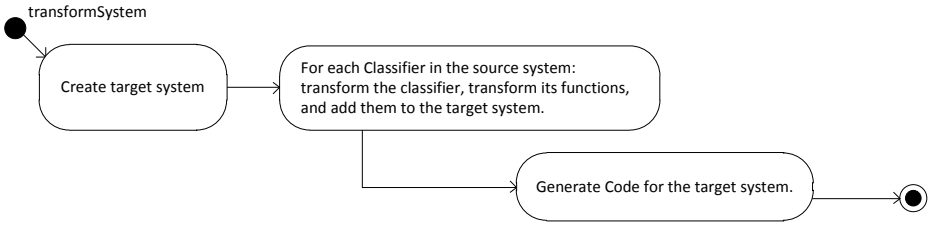

Fig. 3. An activity diagram describing the example transformation

Figure 3 shows an activity diagram which gives an overview of the transformation from a source ASG to a target ASG. At first, the target system has to be created. Each classifier in the source system has to be transformed into a corresponding classifier in the target system. While primitive types can be transformed easily, headers have to be created for all transformed classes. For each method, a function has to be created and a corresponding function declaration has to be added to the correct header. In the end, the target system can be passed to the code generation mechanism.

3 Foundations

In this section, we will briefly introduce story diagrams and their current features. Story diagrams combine UML activity diagrams and graph transformations by embedding graph replacement rules into the activities. This allows the

activities in Figure 3 to be specified formally by graph replacements while preserving the general control flow structure of the example transformation.

In terms of the classification of model transformations proposed by Czarnecki and Helsen [3], story diagrams are an endogenous, in-place transformation language. It has both declarative (pattern matching) and operational elements (specification of control flow): Its control flow allows a deterministic selection of the graph replacement rules to be applied, with a (non-deterministic) pattern matching in the graph replacement rules. It can also be used for inter-model transformations to create a new target model from a given source model, as seen in the example given in this paper. In order to execute story diagrams, code generation [6] and interpretation [8] are supported.

In the following, we will describe the graph transformations, the so-called story patterns, in Section 3.1. Afterwards, we will explain how control flow can be modeled by using elements from activity diagrams in Section 3.2.

3.1 Story Patterns

Story patterns describe graph replacement rules that can be embedded into the activities of a story diagram. They are based on labeled, attributed graphs that are extended by a type model [5]. The types and references that are specified in the type model are used to type the nodes and edges within the story pattern. Type models for story diagrams can be created, e.g., by using EMF Ecore [21]. In our example, we will use the meta-models shown in Figures 1 and 2 as type models. The type model supports inheritance and polymorphism, i.e., a node of type Classifier matches objects of types Classifier, Class, and PrimitiveType. This allows specifying graph replacement rules for object-oriented models.

In order to provide a concise notation, story patterns apply a short-hand notation depicting left-hand side and right-hand side in one graph. Nodes and edges being created (or deleted) are annotated with `<<create>>` (or `<<destroy>>`, respectively). The matching of story patterns in a host graph requires an isomorphic matching of the pattern's left-hand side in the host graph, i.e., two nodes of the pattern may not be matched to the same node in the host graph [5, 19]. The matching is performed with respect to the types of the type model. The deletion of nodes is applied according to the Single Pushout Approach (SPO, [19]), i.e., dangling edges resulting from the deletion of nodes are deleted as well.

Figure 4 shows an example of a story pattern that simply adds a class to the set of classifiers of the source system (cf. Figure 1).

3.2 Control Flow

Story diagrams are an extension of UML 1.4 activity diagrams [16] that embed story patterns into the activities. That allows to model basic control flow structures like branches or loops. Figure 5 shows a story diagram that embeds the story pattern of Figure 4 into one of its activities. The purpose of the story diagram is to create a new class in the source system if no class with the name given by the parameter className already exists.

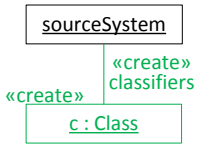

Fig. 4. Simple story pattern

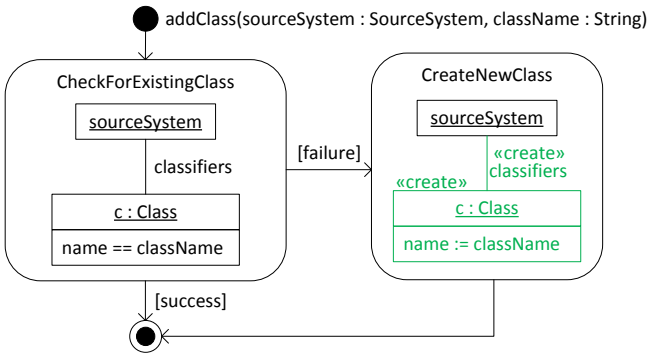

Fig. 5. Control flow in story diagrams

In the first activity, the embedded story pattern tries to bind a class with the respective name in the source system. The sourceSystem is given as a parameter to the story diagram and can be used as a *bound* node by referring to the name of the parameter. A bound node is signified in the concrete syntax by omitting its type information. Then, the story pattern tries to bind an object of type Class to the *unbound* node named c such that the attribute condition is fulfilled. If this pattern can be matched successfully, i.e., the class already exists, the activity is left via the [success] transition and the story diagram terminates. If no such class can be found, the matching fails and the activity is left via the [failure] transition. Then, the second activity creates the class, links it to the source system, and sets the respective name. Additionally, it is possible to add boolean conditions and an [else] to the transitions to model more specific conditions.

In general, an initial matching is established by the parameters. This matching is extended by the story patterns in the activities. Then, the matching is propagated to the next activity along the transitions. If a story pattern fails, the current matching is not changed. In subsequent activities, an object previously bound to a node c can be referenced using a bound node with name c.

The specification of the transformation outlined in Figure 3 can only be accomplished by specifying loops because it requires iterating over all classifiers of the system and all methods of the classes. Loops can be modeled using *forEach activities*. The story patterns in forEach activities are matched as long as new matchings can be found. They are visualized by a double border line as depicted

in Figure 6. The transformation formalizes the informal description of Figure 3 using the current features of story diagrams.

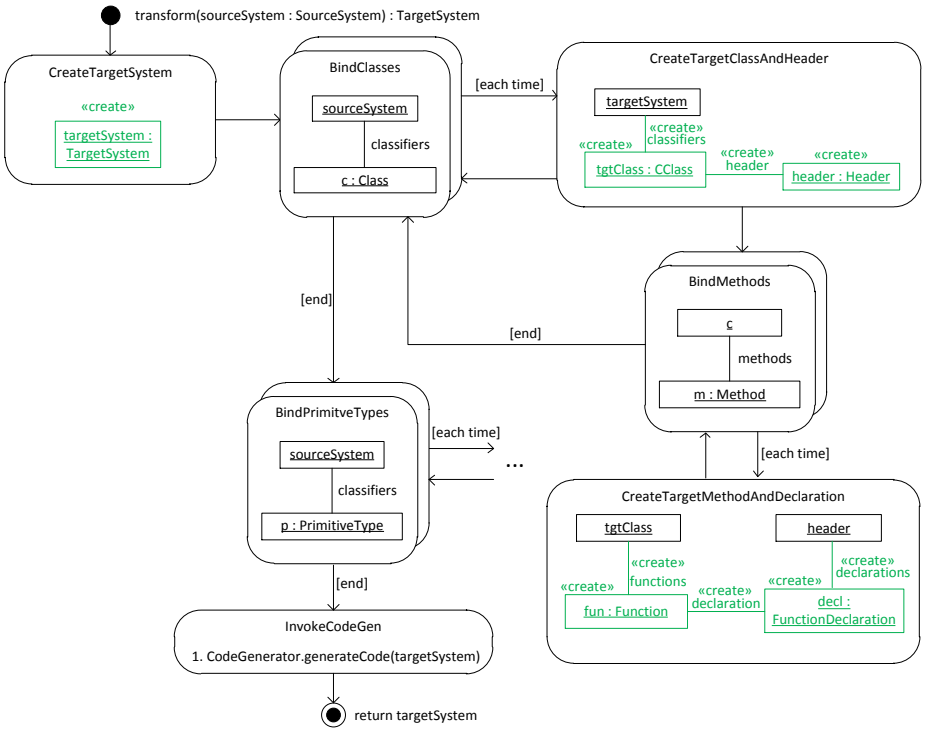

Fig. 6. Example of a complex transformation including several forEach activities

The story diagram creates a target system in its first activity. Then, all classes of the source system sourceSystem are matched. For each match that has been found, the forEach activity is left via the [each time] transition. Then, the third activity creates a respective class and its header in the target system. Afterwards, all methods of the class c of the source system are matched. Again, the forEach activity is left for each new match using the [each time] transition. After all methods have been transformed, the control flow returns to the activity Bind classes to bind the next class. It is required that a control flow that has left a forEach activity eventually returns to this activity to obtain a correct story diagram.

After transforming the classes, all primitive types of the source model have to be transformed. We omit the details here due to space limitations. Finally, the target system object targetSystem is returned by the story diagram as indicated by the annotation return targetSystem on the stop node.

Hitherto, story diagrams only support proprietary calls to library functions called collaboration statements. In Figure 6, the activity InvokeCodeGen contains such a call to the code generator. The call only is a string expression that does neither allow type checking of the input parameters nor getting a matching of return values, i.e., the return values cannot be used in the story diagram.

4 Calls in Story Diagrams

Our concept introduces two variants of calls to story diagrams: *Story diagram calls* and *opaque calls*. Story diagram calls represent the invocation of other story diagrams at some point in the control flow of a given story diagram. This is done by inserting a special activity, the *Story Diagram Call*. In other cases, it may be necessary or more convenient to reuse existing functionality, e.g., from a library. Because these are external operations which are not modeled as story diagrams, we refer to these calls as opaque calls. Both types of calls share common concepts like in- and out-parameters and are described in detail in Section 4.1. Story diagram calls allow for the polymorphic dispatching of calls based on the concrete types of their arguments. This is explained in Section 4.2.

4.1 General properties of calls

The introduction of calls to story diagrams requires an extension of the existing meta-model which we explain in this section. Afterwards, we show how calls are integrated into the example transformation from Figure 6.

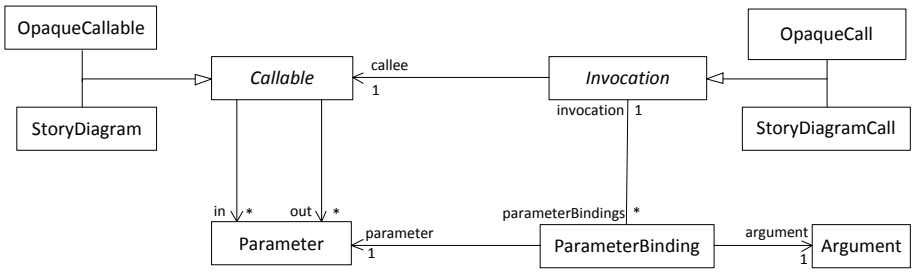

Fig. 7. The meta-model for calls in story diagrams

Figure 7 gives an overview of the (simplified) meta-model which realizes calls in story diagrams. The two elements which can be invoked in story diagrams (other StoryDiagrams and OpaqueCallables) share the common super class Callable. A Callable can be invoked by an Invocation which is either a StoryDiagramCall or an OpaqueCall.

A Callable can have arbitrarily many in- and out-parameters. When invoked, a concrete Argument is assigned to each parameter via a ParameterBinding. Within

a story diagram, nodes can be referenced by their name (cf. Section 3). Consequently, if an object is bound to a node named n somewhere in the story diagram, the identifier n can be used to pass this object as an argument to a call.

In contrast to QVT [18], we do not explicitly model inout-parameters. Instead, we allow the same objects which are passed as in-parameters to be also returned as out-parameters.

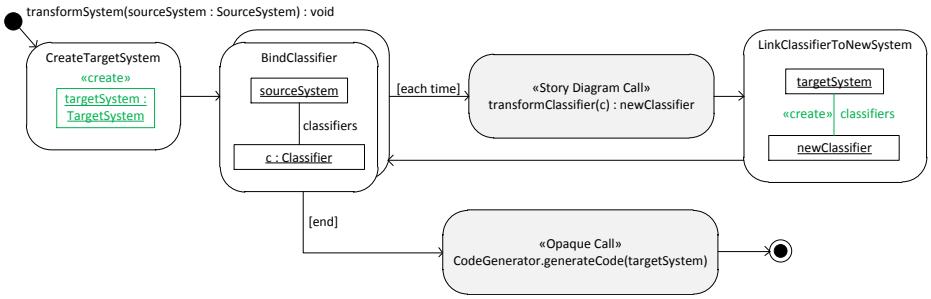

Fig. 8. A story diagram with a story diagram call and an opaque call

The story diagram in Figure 8 realizes the complex transformation depicted in Figure 6 in a concise manner by using call activities. First, a new TargetSystem object is created and bound to the node targetSystem. Then, the activity BindClassifier successively matches all Classifier objects which are connected to the sourceSystem. For each Classifier c, the story diagram transformClassifier is invoked by the Story Diagram Call. The object bound to c is passed as an argument and the returned object of the call is bound to the node newClassifier. This node is reused in the following activity LinkClassifierToNewSystem in which an edge is created between the newClassifier and the targetSystem. In contrast to the story diagram in Figure 6, we can handle all classifiers in a uniform way due to the use of polymorphic dispatching (cf. Section 4.2).

When all classifiers of the sourceSystem have been transformed, the control flow continues via the [end] transition of the BindClassifier activity and the code for the target system can be generated. This is achieved by an Opaque Call to the method CodeGenerator.generateCode. The object bound to the node targetSystem is passed as an argument and no value is returned by the call.

4.2 Polymorphic Dispatching of Calls

The concept presented in this paper also offers the possibility to dispatch the invocation of story diagrams to different story diagrams with matching signatures based on the concrete types of the passed arguments. This is usually referred to as *polymorphic dispatching* or *multiple dispatch*. This allows for more compact

specifications by reducing their conditional complexity. Consider the invocation of transformClassifier in Figure 8. According to the source meta-model from Figure 1, a Classifier can either be a Class or a PrimitiveType. Depending on this, the transformation of the classifier has to be carried out differently.

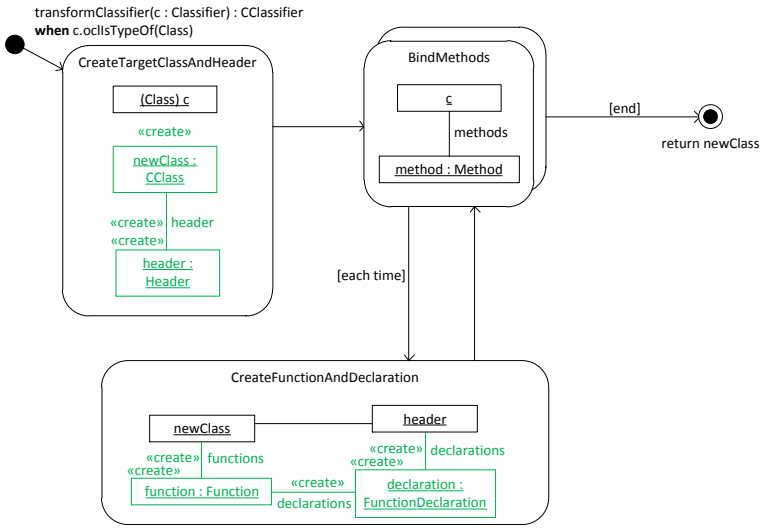

Fig. 9. Story diagram for the transformation of classes

Figures 9 and 10 show two story diagrams for the transformation of classifiers. The story diagram in Figure 9 handles the transformation of a class while the sketched diagram in Figure 10 shows the beginning of the transformation of a primitive data type. The concrete type for which a story diagram is executable is specified by an OCL statement [17] in the when clause underneath the signature. Both transformations have the same signature: transformClassifier(c : Classifier) : CClassifier. The story diagram call in the story diagram in Figure 8 refers to this signature and passes an argument of the common supertype Classifier. At execution time, the call is dispatched to the first story diagram when the argument c is a Class and to the second story diagram when c is a PrimitiveType.

When clauses can concretize the types of all parameters. A story diagram is only executable for a given set of arguments, when the complete when clause is satisfied. Once a story diagram with a matching when clause has been found, it is executed with the given arguments. The when clauses are assumed to specify disjoint conditions so there can only be one suitable story diagram for any given set of arguments. This behavior is similar to the disjoint mapping operations in QVT Operational [18]. If no suitable story diagram can be found, the call cannot be executed and the calling story diagram fails.

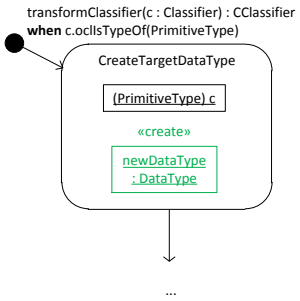

Fig. 10. Story diagram for the transformation of primitive types

In the current version of our dispatching strategy, we restricted ourselves to dispatching based on the type of the parameters. This still allows us to implement a static check ensuring all that all when statements are disjoint and all cases are covered. In the future we could allow arbitrary OCL expressions in when clauses. However, in this case we could only test disjunction and coverage during runtime.

5 Qualitative Evaluation

In our evaluation, we considered a model transformation which was specified with story diagrams. It transformed statecharts into story diagrams to support the analysis of a dynamic communication system in [10]. The communication protocol was given in terms of a statechart while the dynamic changes of the system topology were modeled by in-place model transformations using story diagrams. For an analysis, the statechart was transformed into a graph representing the states and several story diagrams executing the transitions. This transformation was extended to real-time statecharts [7], an extension of statecharts by features from timed automata [1], to support the analysis introduced in [9].

The transformation consists of 20 story diagrams with a total of 147 activities containing 620 nodes. The core of the transformation, however, is contained in four story diagrams translating the states and transitions. These four main story diagrams contain 40 activities with 330 nodes. Due to their size, they are very hard to understand and, what is more, they contain a lot of duplicate functionality. Figure 11 shows an excerpt from one of these story diagrams. In the activity, there exist four nodes on the left side that were created in the preceding activity. The purpose of the activity is to create six edges which are represented by the green UMLLink nodes marked with ≪create≫ and shall connect the four nodes. Obviously, the repeated creation of such nodes can be moved to a helper function by using calls.

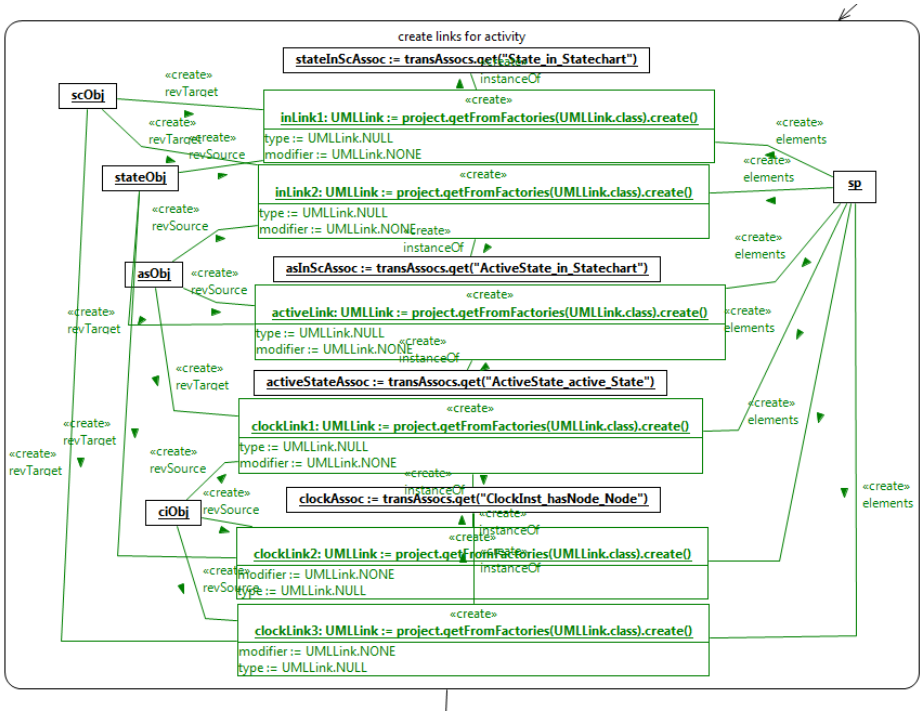

Fig. 11. Complex story diagram before using calls

We performed a qualitative evaluation on how this complex transformation can be restructured using the calls concept introduced in this paper. Table 1 summarizes the results of this restructuring. It compares the numbers of story diagrams, story activities, calls activities, nodes, and the ratio of nodes per activity before and after introducing calls.

Table 1. Reduction of the rule sizes

	# without calls		# with calls	
	main story diagrams	total	main story diagrams	total
Story Diagrams	4	20	4	19 + 7
Story Activities	40	147	28	145
Call Activities	0	0	78	78
Nodes	330	620	74	444
Ø Nodes per story activity	8.25	4.22	2.64	3.06

In detail, we restructured the transformation by the following steps. Initially, one story diagram with five activities containing a total of 20 nodes became

obsolete immediately without any structural changes to the transformations as such. Additionally, the four main transformation rules can be modularized such that we obtain smaller transformations which are easier to create and to maintain. From the main transformations rules, we can move 12 activities containing 85 nodes to four new story diagrams which are then called by the main transformations.

The use of transformation calls also enables us to introduce helper transformations for the repeated creation of objects of the same kind. We identified three possible helper transformations for the creation of objects, links, and attribute assignments in the target model. Figure 12 shows how calls to these helper functions can be utilized to reduce the visual complexity of the transformation depicted in Figure 11. They enable us to remove 171 nodes from the story diagrams and to replace them by 74 calls to these helper functions.

After all these modifications, there is no duplicate functionality in the story diagrams. The number of used objects in the four main transformation rules can be decreased from 330 to 74 by introducing another 78 calls.

In summary, the usage of calls reduces the visual complexity of the main transformation rules by reducing the average number of objects per activity from 8.25 objects to 2.64.

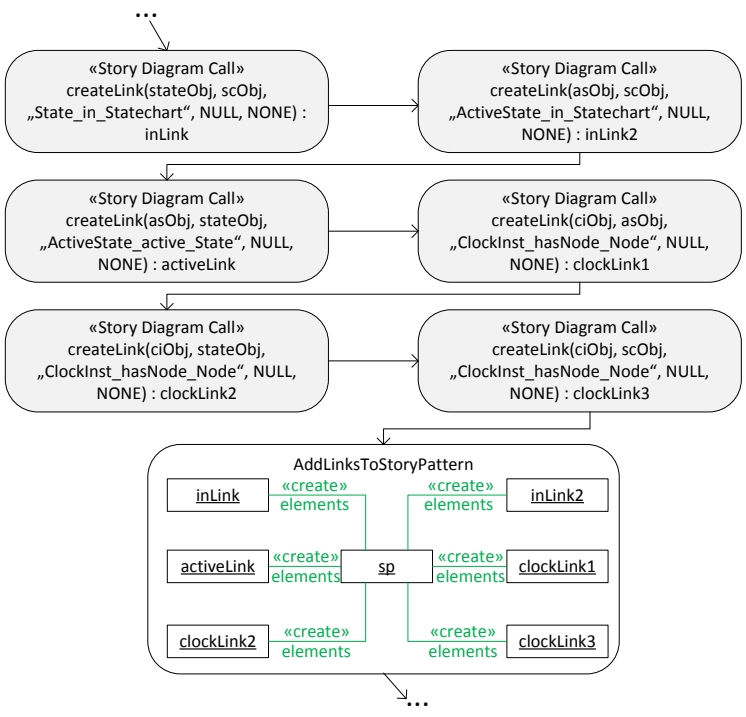

Fig. 12. Complex story diagram after using calls

6 Related Work

Model transformation has become an important research topic during the last years. Several concepts and tools with different scopes and applications have been proposed.

Several model transformation approaches exist which are similar to story diagrams. Here, we focus on those solutions that have a reasonable documentation available. For a more comprehensive overview of transformation approaches see, for example, [3]. Current transformation tools can, for instance, be found in [14].

Henshin [2] is a model transformation language for in-place transformations of EMF-based models. It uses pattern-based rewrite rules (called "transformation rules") and control-flow-based operational semantics (called "transformation units") on top of it. Transformation units can also be called by other transformation units, also including parameters. *MOLA* [12] is an in-place model transformation language with graphical syntax similar to story diagrams. Transformation rules may consist of multiple matching and modification patterns, and the control flow inside a transformation rule can be specified, with a focus on the loop construct. Furthermore, it also allows calling other transformations rules. *VIATRA* [22], a textual language, uses abstract state machines to specify the control flow and graph transformation rules for elementary model manipulations. It also addresses modularization by reusable patterns that are called from the graph transformation rules.

Although most of the story-diagram-like transformation languages include means for specifying control flow (including calling other transformation units), none of them supports polymorphic dispatching. (Note that in most cases polymorphic dispatching can be emulated using other means, but doing so would result in a more complex and less maintainable rule set.)

However, looking at other transformation language types, there are some approaches that support polymorphic dispatching. For instance, *Xpand* [4], a model-to-text transformation language based on templates, uses "polymorphic template invocation" where the most specialized template available is used. However, it only supports single dispatch, i.e., only one parameter is used to determine the used template.

Considering exogenous, inter-model transformations, *QVT Operational* [18] is an operational language that also allows polymorphic dispatching by its disjuncts keyword. A QVT-O mapping operation can declare that a call to it should be dispatched to other mappings. In this case, the invocation of that mapping operation results in the execution of the first mapping whose signature fits the concrete parameters and whose when clause evaluates to true. This is a more powerful construct than our solution, as it not only allows dispatching based on the actual type, but arbitrary constraints. However, there must be a base rule where all dispatching possibilities are listed; in our solution, all signature-compatible transformations with the same name are automatically used in dispatching, allowing a better modularization as well as an easier extension of the rule set. Furthermore, QVT-O is a textual transformation language which may not be well-suited in many cases [15].

In declarative inter-model transformation languages like *Triple Graph Grammars* (TGGs) [20], the control flow cannot be defined explicitly. Instead, the order of the rule application is implicitly defined by preconditions of the transformation rules. However, when more than one rule has a fitting precondition, the rule to be applied is selected non-deterministically, dependent on the concrete transformation tool implementation, or by a given rule priority. Klar et al. [13] proposed a rule generalization concept with a precedence for the most refined rules, a solution similar to polymorphic dispatching. In *QVT Relations* [18], which is similar to TGGs, control flow may also be explicitly specified by using where clauses.

The *Atlas Transformation Language* (ATL) [11] is a hybrid inter-model transformation language, integrating declarative and operational aspects. It is similar to QVT, but only has a textual representation of the transformation rules.

7 Conclusions and Future Work

The specification of control flow has always been an integral part of story diagrams. However, story diagrams did not support structuring complex transformations into several independent story diagrams up to now. In this paper, we described an extension of the existing formalism for the control flow to allow the invocation of other story diagrams, including parameters and return values. Furthermore, we introduced a concept to polymorphically dispatch a call to the story diagram which matches the actual run-time types of the bound parameters.

By this means, story diagrams become more flexible and reusable and allow a better structuring of the rule set. In a qualitative evaluation, we have discussed that these new features can help to reduce the size of the rule set and increase its maintainability.

As future work, we plan to investigate whether and how the polymorphic dispatching can be extended, for example, to allow arbitrary dispatching conditions similar to QVT-O. We also plan to evaluate the maintainability, conciseness, and understandability of our approach in comparison to other model transformation languages in an empirical study.

References

1. R. Alur. Timed Automata. In *Proceedings of the 11th International Conference on Computer Aided Verification (CAV '99)*, volume 1633 of *Lecture Notes in Computer Science (LNCS)*, pages 8–22. Springer, 1999.
2. E. Biermann, C. Ermel, J. Schmidt, and A. Warning. Visual Modeling of Controlled EMF Model Transformation using Henshin. In *Proceedings of the 4th International Workshop on Graph-Based Tools (GraBaTs 2010)*, 2010.
3. K. Czarnecki and S. Helsen. Feature-based Survey of Model Transformation Approaches. *IBM Systems Journal*, 45:621–645, July 2006.
4. Eclipse Modeling Project. Model To Text (M2T) – Xpand. http://www.eclipse.org/modeling/m2t/?project=xpand.

5. T. Fischer, J. Niere, L. Torunski, and A. Zündorf. Story Diagrams: A New Graph Rewrite Language Based on the Unified Modeling Language and Java. In *TAGT '98 Selected papers*, volume 1764 of *Lecture Notes in Computer Science (LNCS)*, pages 296–309. Springer, 2000.
6. L. Geiger, T. Buchmann, and A. Dotor. EMF Code Generation with Fujaba. In *Proceedings of the 5th International Fujaba Days*, 2007.
7. H. Giese and S. Burmester. Real-Time Statechart Semantics. Technical Report tr-ri-03-239, Software Engineering Group, University of Paderborn, Germany, 2003.
8. H. Giese, S. Hildebrandt, and A. Seibel. Improved Flexibility and Scalability by Interpreting Story Diagrams. In *Proceedings of the 8th International Workshop on Graph Transformation and Visual Modeling Techniques (GT-VMT '09)*, volume 18 of *Electronic Communications of the EASST*, 2009.
9. C. Heinzemann, J. Suck, and T. Eckardt. Reachability Analysis on Timed Graph Transformation Systems. In *Proceedings of the 4th International Workshop on Graph-Based Tools (GraBaTs 2010)*, 2010.
10. C. Heinzemann, J. Suck, R. Jubeh, and A. Zündorf. Topology Analysis of Car Platoons Merge with FujabaRT & TimedStoryCharts - a Case Study. In *Transformation Tool Contest 2010*, 2010.
11. F. Jouault, F. Allilaire, J. Bézivin, and I. Kurtev. Atl: A model transformation tool. *Science of Computer Programming*, 72(1-2):31 – 39, 2008.
12. A. Kalnins, J. Barzdins, and E. Celms. Model Transformation Language MOLA. In *Proceedings of Model-Driven Architecture: Foundations and Applications (MDAFA) 2004*, 2004.
13. F. Klar, A. Königs, and A. Schürr. Model Transformation in the Large. In *Proceedings of the the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC-FSE '07)*, pages 285–294. ACM, 2007.
14. S. Mazanek, A. Rensink, and P. van Gorp, editors. *Transformation Tool Contest 2010*, Malaga, Spain, 2010.
15. D. L. Moody. The "Physics" of Notations: Toward a Scientific Basis for Constructing Visual Notations in Software Engineering. *Software Engineering, IEEE Transactions on*, 35(6):756 –779, 2009.
16. Object Management Group. Unified Modeling Language (UML), Specification, Version 1.4, 2001. OMG document formal/2001-09-67.
17. Object Management Group. Object Constraint Language (OCL), Version 2.2, 2010. OMG document formal/2010-02-01.
18. Object Management Group. Query/View/Transformation (QVT), Version 1.1, 2011. OMG document formal/2011-01-01.
19. G. Rozenberg, editor. *Handbook of Graph Grammars and Computing by Graph Transformation: Vol. I. Foundations*. World Scientific Publishing Co., Inc., 1997.
20. A. Schürr. Specification of Graph Translators with Triple Graph Grammars. In *20th International Workshop on Graph-Theoretic Concepts in Computer Science*, volume 903 of *Lecture Notes in Computer Science (LNCS)*, pages 151–163. Springer, 1994.
21. D. Steinberg, F. Budinsky, M. Paternostro, and E. Merks. *EMF: Eclipse Modeling Framework*. The Eclipse Series. Addison-Wesley, 2nd edition, 2008.
22. D. Varró and A. Balogh. The model transformation language of the viatra2 framework. *Science of Computer Programming*, 68(3):214 – 234, 2007. Special Issue on Model Transformation.