Feature Report: Modeling and Interpreting EMF-based Story Diagrams

Holger Giese
Hasso-Plattner-Institute at the University of Potsdam
Prof.-Dr.-Helmert-Str. 2-3
14482, Potsdam, Germany
holger.giese@hpi.uni-potsdam.de

Stephan Hildebrandt
Hasso-Plattner-Institute at the University of Potsdam
Prof.-Dr.-Helmert-Str. 2-3
14482, Potsdam, Germany
stephan.hildebrandt@hpi.uni-potsdam.de

Andreas Seibel
Hasso-Plattner-Institute at the University of Potsdam
Prof.-Dr.-Helmert-Str. 2-3
14482, Potsdam, Germany
andreas.seibel@hpi.uni-potsdam.de

ABSTRACT

In this paper, we report on the current state of development of our tools around modeling and interpreting EMF-based Story Diagrams, which comprises a graphical editor and an interpreter for Story Diagrams. The editor provides useful features like model validation for Story Diagrams, and advanced editing features like syntax highlighting and code completion for OCL expressions. The interpreter was initially presented in [5]. We showed that the interpreter enables new possibilities, improves the flexibility of applying Story Diagrams but also can improve the performance of execution Story Diagrams. In the meantime, the interpreter evolved. Beside its basic features of a dynamic pattern matching strategy and compatibility to dynamic EMF, we introduce new features like support for map-typed references, containment links and a notification mechanism.

Categories and Subject Descriptors

D.2.m [**Software Engineering**]: Miscellaneous; H.1.m [**Models and Principles**]: Miscellaneous

Keywords

Story Diagram, Interpreter

1. INTRODUCTION

Story Diagrams were introduced in [2]. They combine UML Activity Diagrams with graph transformation rules to graphically describe the search for patterns in an object graph and the creation and deletion of objects and links. They were initially implemented into the Fujaba CASE tool [8], which supports modeling Story Diagrams and generating executable code from Story Diagrams.

In the past years, the Eclipse Modeling Framework (EMF)[1] has become the industry standard in the area of model-driven software engineering based on the Eclipse platform. There are many tools supporting model-driven software development (e.g., openArchitectureWare[2]), model transformation (e.g., ATL[3]) or other activities regarding models that are based on EMF. This has lead to the situation that we need to support EMF in current projects in order to be compatible to existing tools. However, we still need Fujaba to model Story Diagrams and generate executable code for further application. Although there are attempts to minimize this technology gap, e.g., an EMF-compatible code generator [3] or an EMF adapter for Fujaba models [7], it still poses an obstacle as we already outlined in [1] in more detail.

Another central drawback with Story Diagrams in Fujaba is the static pattern matching strategy, which can be a serious performance bottleneck. This strategy is determined at generation time based solely on the information available in the meta models. The strategy prefers to-one links over to-many links to match objects. However, it does not distinguish between to-many links with different numbers of contained objects. This information is only available at runtime. Therefore, the static pattern matching strategy is the same for all possible instance models.

To overcome this obstacle, we developed an interpreter for Story Diagrams [5] based on Eclipse and EMF.[4] Its main improvement among other features is a dynamic pattern matching strategy. This dynamic pattern matching strategy leverages information available in the instance models and distinguishes between to-many links depending on their exact size. The performance can vary tremendously between dynamic and static pattern matching strategies. In [5] the results of a benchmark are presented, which show the effect of different pattern matching strategies concerning the performance of the pattern matching process. The benchmarks showed that Fujaba-generated Story Diagrams with non-optimal pattern matching strategies are much slower than the interpreter.

[1]http://www.eclipse.org/modeling/emf/

[2]http://www.openarchitectureware.org

[3]http://www.eclipse.org/m2m/atl/

[4]The interpreter and editor for Story Diagrams can be downloaded from our update site: http://www.hpi.uni-potsdam.de/giese/gforge/sam-update-site/site.xml. Prerequisites are Eclipse 3.5, EMF 2.5, GMF 2.2, Xtend 0.7 and OCL 1.3.

Another promising feature of the interpreter is the compatibility to dynamic EMF. Dynamic EMF omits generated code. Instead, dynamic EMF objects are instances of a common class, *DynamicEObjectImpl*. These instances can be populated with data from, e.g. an XMI file or a database. Accessing this data is only possible via the reflective EMF API, which is also implemented by generated code. While the interpreter uses only this API, it is completely transparent whether dynamic objects or generated code is used. Because no code generation of the meta models is required, meta models can be exchanged/modified easily and the code that uses it does not need to be adapted. Another interesting observation of the performance evaluation is, that there is virtually no difference in execution times if the interpreter works on dynamic EMF objects compared to using generated code.

The interpreter is already used in several projects at our research group. Among them is a model transformation system based on Triple Graph Grammars [4] and a model-driven configuration management system [6]. The model-driven configuration management system uses the interpreter extensively, e.g., Story Diagrams are used to generate Story Diagrams. Thus, we are able to configure Story Diagrams at runtime, generate the configured Story Diagrams and instantaneously execute the configured Story Diagrams with the interpreter.

In this paper, we report on additional features that come with the new version of the Story Diagram interpreter and editor. Both provide an extension mechanism that can be used to integrate interpreters and editing features for additional expression languages. Expression languages are textual languages to express constraints or actions. At the moment, OCL is the only supported expression language. OCL expressions in a Story Diagram are evaluated by the interpreter and the editor is extended with code completion and syntax highlighting for OCL. Furthermore, the interpreter supports map-typed references to overcome the lack of qualified associations known from Fujaba, as well as containment links to match objects that are a direct or indirect part of a container. Additionally, the interpreter supports notifications, which enables new possibilities like debugging but also leveraging incremental application of Story Diagrams. The graphical editor for Story Diagrams was enhanced, too. Besides providing basic modeling support for Story Diagrams, the editor supports modeling all new features of the interpreter. It further comes with a validation integration to check the syntactic and, partly, semantic correctness of Story Diagrams.

In the sequel of this paper, we briefly introduce the interpreter and subsequently explain all new features of the interpreter in Section 2. In Section 3, the Story Diagram editor and its features are described. The paper closes with an outlook on future work in Section 4, especially on the features that are currently missing.

2. STORY DIAGRAM INTERPRETER

The Story Diagram interpreter consists of several components: The *interpreter* itself, which is responsible for traversing the overall Story Diagram starting from the initial node and invoking the other components if necessary; the *variables manager*, which is the central store of variables and their values; additional *interpreters for expression languages*, that can be plugged into the Story Diagram interpreter via an extension mechanism to allow execution of expressions written in that language; and the *Story Pattern Matcher*, that executes a single Story Action Node. The pattern matcher tries to find matches for the Story Pattern using a dynamic pattern matching strategy, and creates and deletes objects if a match was found. The matching process is explained in detail in [5].

2.1 Enhancing Story Patterns

A special feature of the Story Pattern Matcher is the analysis phase preceding the execution. Its purpose is to sort the links and objects and introduce new links into the Story Pattern to make the pattern matching process even more efficient.

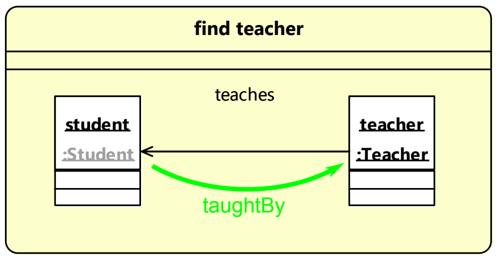


Figure 1: Another link is inserted that represents the opposite link of a bidirectional reference.

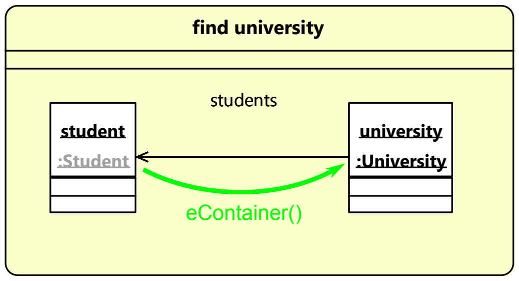


Figure 2: Another link representing the *eContainer()* call is inserted.

The introduction of new links is shown in Fig. 1. The *student* is a bound object (indicated by the grayed out type of the object). The pattern matching starts here to find matches for the other object. The *teaches* link leads from the teacher to the student but the reference in the meta model is a bidirectional reference. The interpreter notices that and creates an opposite link.[5] The same is done for containment references (see Fig. 2). EMF provides the *eContainer()* operation to get the container object of an object. The *students* reference is a unidirectional containment reference. Therefore, the *university* can be easily matched by calling *eContainer()* on the student. The inserted link represents this call.

[5]Of course, the modeler can explicitely model both directions but the analysis phase makes this unnecessary.

2.2 Expression Languages

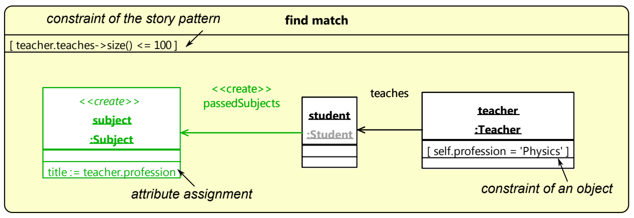
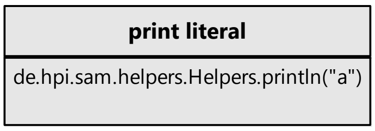


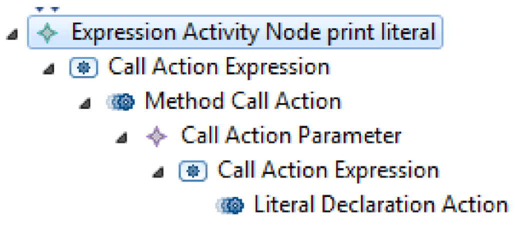
Figure 3: Expressions are used in many places to express constraints, queries and actions.

There are many places where it is desirable to use textual languages to express a constraint, a query or an action. The interpreter and the Story Diagram editor provide an extension mechanism that allows other plug-ins to integrate textual expression languages. These plug-ins must provide at least an interpreter for their expression language. In addition, they can provide a custom *Source Viewer* implementation, that includes features like syntax highlighting and code completion (see Section 3.2 for more information). Support for OCL expressions is already included.

Fig. 3 shows several examples where constraints can be used. If they are used on a Story Pattern object (e.g., the *teacher*), the constraint is evaluated as soon as a possible match was found. This implies that other Story Pattern objects might not be matched at this time. Therefore, references to them should not be used in such constraints. Instead, these constraints should be attached to the overall Story Pattern, so they are evaluated after a match for the whole pattern was found. If used as constraints, expressions have to return a boolean value. However, they can also be used as queries to return an arbitrary value. This is the case with the attribute assignment of the *subject*. The expression is evaluated and its value is assigned to the *title* attribute. Within an expression, all created variables of the Story Diagram are available, e.g. the *teacher* in the attribute assignment of *subject*.



(a) An Expression Activity node that calls an external helper operation.



(b) Abstract syntax of the expression in Fig. 4(a)

Another place are Expression Activity Nodes (see Fig. 4(a)), which are comparable to Fujaba's Statement Activities. They contain an expression, that is executed but whose return value is ignored. It is intended to use expressions with side effects here to perform arbitrary actions.

While OCL is free of side effects and the integration of existing scripting languages, like Lua, PERL, Python, etc. poses some difficulties, we decided to integrate another kind of expression language where the expressions are directly modeled as trees instead of a text string. Fig. 4(b) shows an example. These *Call Action Expressions* provide basic functionality to declare literals and variables, reference variables, create new objects and, most importantly, invoke arbitrary Java methods via reflection. *Call Action Expressions* can be nested and mixed with string expressions. In Fig. 4(b) a *Method Call Action* is used to invoke the method. The parameter *"a"* is provided by a *Literal Declaration Action*. A major advantage of *Call Action Expressions* is, that they can be evaluated very quickly. It is not necessary to parse a string. In contrast, using OCL expressions has a notable impact on the performance.

2.3 Map-Typed References

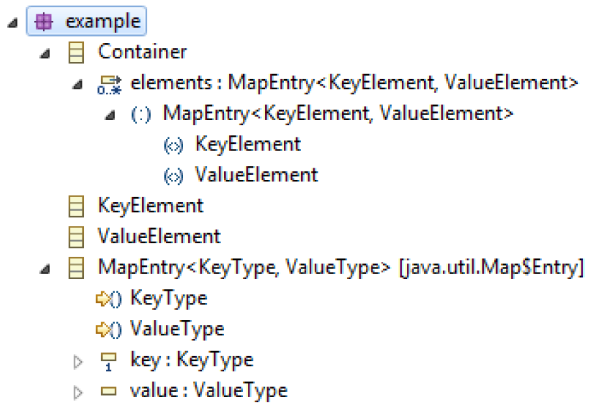

example
Container
elements : MapEntry<KeyElement, ValueElement>
MapEntry<KeyElement, ValueElement>
KeyElement
ValueElement
KeyElement
ValueElement
MapEntry<KeyType, ValueType> [java.util.Map$Entry]
KeyType
ValueType
key : KeyType
value : ValueType

Figure 4: A meta model containing a map entry.

Sometimes, mappings from keys to values are required to model qualified associations, for example. EMF provides so-called *Map-Typed References*, references to map entries, which are parameterized with key and value types. These references allow mapping keys to values. This is shown in Fig. 4. The class *Container* has a map, that maps *KeyElements* to *ValueElements*. To realize this, a helper class has to be created. Its instance type name must be *java.util.Map$Entry*. It has a *key* and a *value* attribute and generic type parameters. The *elements* reference of the *container* is a to-many containment reference to the *MapEntry*, which is parameterized with the *KeyElement* and *ValueElement* types. EMF's code generator recognizes this pattern and creates a map instead of a list.

This can be modeled in the same way in a Story Pattern, which is shown in Fig. 5. The map entry is a distinct object in the Story Pattern. The *value* object can be easily obtained by querying the map because the *key* object is already bound, here. However, now the interpreter does not exploit the fact that this is a map. The interpreter rather treats it as a list of map entries, i.e. it examines all map entries and checks for each entry whether its key is the *key* object. Furthermore, the *MapEntry* object is not relevant to the modeler of the Story Diagram. It rather makes the diagram more complicated. Therefore, we decided to introduce a special kind of link, a *Map Entry Story Pattern Link*.

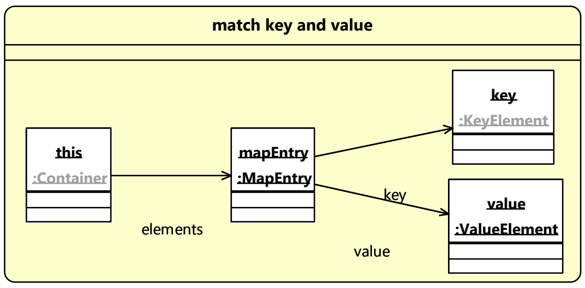


Figure 5: Modeling a map entry as a distinct object in the Story Pattern.

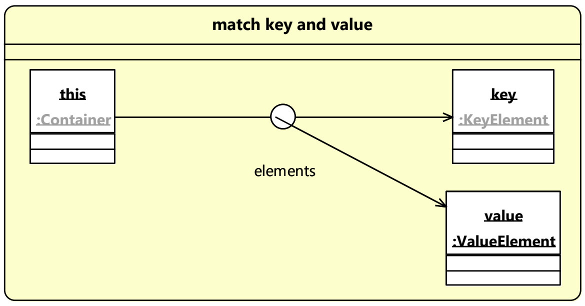


Figure 6: Modeling a map entry as a *Map Entry Story Pattern Link*.

Fig. 6 shows the same case like Fig. 5 but this time the map entry is hidden by a *Map Entry Story Pattern Link*. When this type of link is used, the interpreter will exploit the fact, that there is a map and simply query it for the value belonging to the *key* object. Of course, this is not the best possible solution. A better way would be if EMF would provide a more elegant way to model maps.

2.4 Containment Links

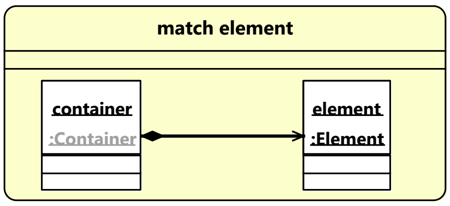


Figure 7: A Containment Link indicates that the *Element* should be contained directly or indirectly in the *Container*.

Ordinary Story Pattern links specify which reference from the meta model they are representing. So the interpreter only needs to follow the corresponding instance links to find matches for unbound objects. However, sometimes it is not known how the unbound objects are connected to a bound root object. It may be unknown, via which reference they are connected or how many levels of indirection are in between. If it is at least known that objects are connected via containment associations, *Containment Links* (see Fig. 7 can be used to model this case. The *Containment Link* indicates that the *Element* is directly or indirectly contained in the *Container*. This feature can be implemented very easily by using the *eAllContents()* operation of EMF's reflective API. This operation returns a tree iterator, that returns all direct and indirect children. The opposite direction is also possible, i.e. the element is bound and a match for the container is sought. In this case, the interpreter climbs the containment hierarchy upwards by calling *eContainer()* repeatedly until a match was found or the top container was reached.

2.5 Interpreter Notifications

The interpreter provides a notification mechanism, which is based on EMF's notification mechanism. By attaching a notification listener to the interpreter and setting a debug flag, notifications about all relevant execution steps can be received. This allows for example to implement a visual debugger for Story Diagrams. In the model-driven configuration management system [6], notifications are leveraged for incremental Story Diagram application. In case that a Story Diagram successfully matched, the traversed objects that lead to the successful application are stored in a map together with the model of the Story Diagran. In case that objects change, we can easily determine which Story Diagrams have to be re-evaluated.

3. STORY DIAGRAM EDITOR

In addition to the interpreter itself, we also built a graphical editor for Story Diagrams based on the Graphical Modeling Framework (GMF)[6]. Using GMF, graphical editors for EMF-based models can be created with comparably little effort. It also provides basic features like Copy&Paste, printing or layouting.

3.1 Validation

An important feature of an editor is the ability to check a model for correctness to indicate modeling errors to the user. EMF provides an extensible validation mechanism. Plug-ins can integrate into this validation mechanism by extending some extension points. They can register a validator for a specific meta model. If an instance of that meta model is validated, the validator is invoked. The validation can be triggered in the editor (both the tree editor generated by EMF, and the graphical editor based on GMF) or from Java code. Unfortunately, the validator has to be implemented in Java, which makes specifying simple validation rules rather cumbersome.

```
context StoryPatternObject ERROR
  "A story pattern object must have a name.":
  this.name != null && this.name.trim() != "";
```

Figure 8: Example constraint that checks if the name of a Story Pattern object is set.

To ease the definition of validation rules, we use Xtend's Check language which is a constraint language similar to OCL. Fig. 8 shows a constraint that checks if the name

[6]http://www.eclipse.org/gmf/

of a Story Pattern is set. The Xtend plug-in provides a generic validator that can be used to execute Check constraints which are declared in a text file. This allows to define the constraints that should hold on a valid Story Diagram in a concise and easily maintainable way. An exception is the validation of textual expression because this requires a more thorough analysis of the Story Diagram. For example, in order to check an OCL expression, the names and types of all available variables must be acquired first. This is very difficult to express in Check.

3.2 Advanced Editing Features

As pointed out in Section 2.2, plug-ins can provide support for other expression languages, including support for advanced editing features in the editor. This is especially useful for users that are unfamiliar with a certain language or if many variables are used in the Story Diagram and a code completion feature lists all available variables.

Technically, this is realized by providing custom implementations of *org.eclipse.jface.text.source.ISourceViewer* and implementing a small interface that allows the editor to access this custom *ISourceViewer* and get and set the text of the viewer. A custom implementation allows to provide syntax highlighting and other visual annotations, code completion and content assist features. For OCL, we provide basic syntax highlighting and code completion. If a plug-in does not provide an own implementation, a default source viewer is used. Of course, advanced editing features are not available in this case.

4. OUTLOOK

The interpreter solves our two most urgent problems. The dynamic pattern matching strategy leads to a better average and worst-case performance. The interpreter offers a tight integration with EMF and is compatible to dynamic EMF, which improves the flexibility of Story Diagrams and their application. The additional features of the interpreter in combination with the graphical editor and its features improve the usability of Story Diagrams within Eclipse, especially for people unfamiliar with Story Diagrams.

However, there are still limitations and space for improvements. A major disadvantage of the interpreter, compared to Fujaba, is the lack of integrating arbitrary Java code directly into Story Diagrams, because no interpreter is available. Currently, we employ expressions languages to overcome this drawback. These are either textual languages or basic tree based models (*Call Action Expressions*, see Section 2.2), that can be used to express conditions and queries, and perform actions. However, OCL is the only supported textual language, yet. Another limitation is that Story Diagrams need to be link into an operation definition of a class of a meta model. This limits the dynamic capabilities because the operation signatures must be defined a-priory and are thus fixed. Therefore, we plan a less strict concept. The idea is to hand over a dynamic start-graph, which does not need to fit into the signature of the related method.

In addition we plan to extend the interpreter with basic features, known from Fujaba. For example, negative application conditions are not supported, yet. Instead, OCL constraints have to be used for this purpose. Other missing features are path expressions, optional objects, sets of objects and syntax checks of OCL expressions. Another problem is the mentioned impact of evaluating OCL expressions on the performance. A possible solution could be to parse the expression beforehand and store the expression's abstract syntax tree in the Story Diagram.

Acknowledgements

We like to thank all students that helped implementing and improving the Story Diagram interpreter and the graphical editor. These are in particular Mark Liebetrau, Sebastian Wätzoldt and Christian Lück.

5. REFERENCES

[1] B. Becker, H. Giese, S. Hildebrandt, and A. Seibel. Fujaba's Future in the MDA Jungle - Fully Integrating Fujaba and the Eclipse Modeling Framework? In *Proceedings of the 6th International Fujaba Days*, 2008.

[2] T. Fischer, J. Niere, L. Torunski, and A. Zündorf. Story Diagrams: A new Graph Rewrite Language based on the Unified Modeling Language. In G. Engels and G. Rozenberg, editors, *Proc. of the 6th International Workshop on Theory and Application of Graph Transformation (TAGT), Paderborn, Germany*, LNCS 1764, pages 296–309. Springer Verlag, 1998.

[3] L. Geiger, T. Buchmann, and A. Dotor. EMF Code Generation with Fujaba. In *5th International Fujaba Days*, Kassel, Germany, October 2007.

[4] H. Giese and S. Hildebrandt. Efficient Model Synchronization of Large-Scale Models. Technical Report 28, Hasso Plattner Institute at the University of Potsdam, 2009.

[5] H. Giese, S. Hildebrandt, and A. Seibel. Improved Flexibility and Scalability by Interpreting Story Diagrams. In T. Magaria, J. Padberg, and G. Taentzer, editors, *Proceedings of the Eighth International Workshop on Graph Transformation and Visual Modeling Techniques (GT-VMT 2009)*, 2009.

[6] H. Giese, A. Seibel, and T. Vogel. A Model-Driven Configuration Management System for Advanced IT Service Management. In *Proceedings of the 4th International Workshop on Models@run.time at the 12th IEEE/ACM International Conference on Model Driven Engineering Languages and Systems (MODELS 2009)*, Denver, Colorado, USA, October 2009. accepted.

[7] J. Johannes. Letting EMF Tools Talk to Fujaba through Adapters. In *Proceedings of the 6th International Fujaba Days 2008*, 2008.

[8] T. Klein, U. A. Nickel, J. Niere, and A. Zündorf. From UML to Java And Back Again. Technical Report tr-ri-00-216, University of Paderborn, Paderborn, Germany, 1999.