An Adaptable Tool Environment for High-level Differencing of Textual Models

[1]Timo Kehrer, Christopher Pietsch, Udo Kelter
[2]Daniel Strüber, Steffen Vaupel

[1]University of Siegen, Germany
{kehrer,cpietsch,kelter}@informatik.uni-siegen.de
[2]Philipps-Universität Marburg, Germany
{strueber,svaupel}@informatik.uni-marburg.de

Abstract. The use of textual domain-specific modeling languages is an important trend in model-driven software engineering. Just like any other primary development artifact, textual models are subject to continuous change and evolve heavily over time. Consequently, MDE tool chain developers and integrators are faced with the task to select and provide appropriate tools supporting the versioning of textual models. In this paper, we present an adaptable tool environment for high-level differencing of textual models which builds on our previous work on structural model versioning. The approach has been implemented within the SiLift framework and is fully integrated with the Xtext language development framework. We illustrate the adaptability and practicability of the tool environment using a case study which is based on a textual modeling language for simple web applications.

1 Introduction

Model-driven engineering (MDE) is a software development methodology which has gained a lot of interest in many application domains. Besides the MDA initiative and related standards promoted by the OMG, the use of textual domain-specific modeling languages (DMSLs) has emerged as an important trend in modern MDE. Textual DSMLs typically have a small scope and formalize the key concepts of a particular domain of interest. Just like any other primary development artifact, textual models are subject to continuous change and heavily evolve over time. Consequently, appropriate tools supporting standard versioning tasks are strongly required, the calculation of a difference between versions of a model being the most fundamental service.

Selecting a proper versioning tool environment often leads to cost/benefit considerations: On the one hand, one can use off-the-shelf line-based difference tools. This option is attractive since these tools are generic in the sense that they can operate with any kind of textual documents. However, differences are reported on a low level of abstraction and often fail to report complex model changes in a meaningful way. On the other hand, there are sophisticated approaches to structural differencing and merging whose advantages over the classical line-based proceeding are undisputed [25,12]. However, virtually all of these solutions come at a price: many tool components have to be re-implemented for each modeling language anew. Considering the large number of different DSMLs which have to be supported, this often leads to a prohibitive effort.

In this paper, we present a flexible tool environment for high-level differencing of textual models which can be adapted to a new language with moderate effort. The approach builds on our previous work on structural model versioning [18,17,16], which was motivated and has been developed in the context of visual modeling languages. The typical effort to configure a differencing tool ranges between 1 and 10 days, depending on the size of the meta-model. In this paper, we focus on the technical extensions required to support textual models. We argue that a difference tool which is tailored to a given DSML provides significant improvements over existing line-based difference tools. In particular, complex restructurings on a model can be detected, and changes are therefore reported on a higher level of abstraction.

The approach has been implemented within the SiLift framework [27]. It uses several tool components which are based on the Eclipse Modeling Framework (EMF) [8] and is fully integrated with the Xtext language development framework [30]. We illustrate the practicability and adaptability of the tool environment using a case study which is based on a textual modeling language for simple web applications.

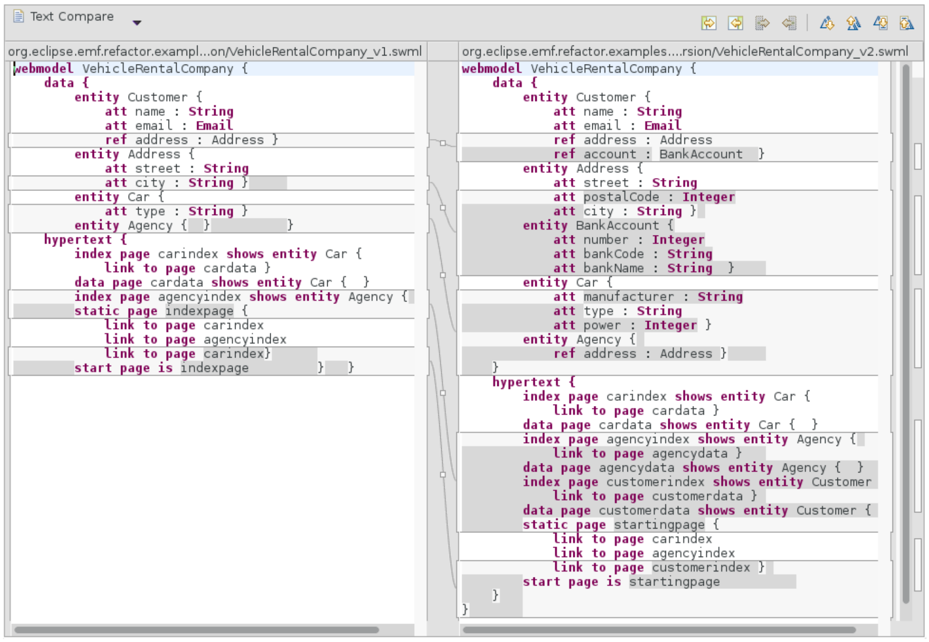
2 Case Study and Motivating Example

In this section, we introduce the case study which will be used to illustrate our approach. The textual modeling language called SWML is introduced in Sec. 2.1. A scenario which describes typical restructurings and improvements on a sample SWML model is described in Sec. 2.2.

2.1 SWML: Simple Web Modeling Language

The Simple Web Modeling Language (SWML) is a textual DSML which aims at defining platform-independent models for a specific kind of web applications. The language has been originally introduced in [7], which also describes a transformation tool chain for generating web applications using standard web development technologies. In this paper, we use the SWML as defined in [5]. In order to keep the paper self-contained, we give an informal description of the SWML abstract syntax:

A WebModel consists of two parts: the DataLayer and the HypertextLayer. The data layer models the application data following basic principles which are known from entity-relationship modeling. An Entity (which is actually an entity type) may have Attributes and References (reference types) to other entity types. Predefined SimpleTypes can be used in attribute declarations. The hypertext layer defines how to present the data using web pages. A Page is either a StaticPage having a fixed content, or a DynamicPage which presents data related to a dedicated entity type. There are two types of dynamic pages: an IndexPage lists the



org.eclipse.emf.refactor.exampl...on/VehicleRentalCompany_v1.swml

```
webmodel VehicleRentalCompany {
    data {
        entity Customer {
            att name : String
            att email : Email
            ref address : Address }
        entity Address {
            att street : String
            att city : String }
        entity Car {
            att type : String }
        entity Agency {  }         }
    hypertext {
        index page carindex shows entity Car {
            link to page cardata }
        data page cardata shows entity Car {  }
        index page agencyindex shows entity Agency {
        static page indexpage {
            link to page carindex
            link to page agencyindex
            link to page carindex}
        start page is indexpage          }   }
```

org.eclipse.emf.refactor.examples....rsion/VehicleRentalCompany_v2.swml

```
webmodel VehicleRentalCompany {
    data {
        entity Customer {
            att name : String
            att email : Email
            ref address : Address
            ref account : BankAccount  }
        entity Address {
            att street : String
            att postalCode : Integer
            att city : String }
        entity BankAccount {
            att number : Integer
            att bankCode : String
            att bankName : String  }
        entity Car {
            att manufacturer : String
            att type : String
            att power : Integer }
        entity Agency {
            ref address : Address }
    }
    hypertext {
        index page carindex shows entity Car {
            link to page cardata }
        data page cardata shows entity Car {  }
        index page agencyindex shows entity Agency {
            link to page agencydata }
        data page agencydata shows entity Agency {  }
        index page customerindex shows entity Customer
            link to page customerdata }
        data page customerdata shows entity Customer {
        static page startingpage {
            link to page carindex
            link to page agencyindex
            link to page customerindex }
        start page is startingpage
    }
}
```

Fig. 1: Initial version v_1 of a sample SWML model and its improved revision v_2

instances of a certain entity type, while a DataPage shows concrete information on a specific entity. The structure of the hypertext layer can be modeled by Links connecting two pages. One of the pages can be declared to be the starting page of the application.

2.2 Example Scenario: Improvements on a Sample SWML Model

SWML models can be conveniently defined using a textual notation. An example model called VehicleRentalCompany taken from [5] is shown in Fig. 1. Version v_1 on the left comes from an early development stage and is used in [5] to illustrate and evaluate quality assurance techniques on textual models. Using metrics and smells as indicators for quality issues concerning the quality aspect completeness, model version v_1 is improved to become version v_2 on the right-hand side of Fig. 1 by applying the following refactorings and manual changes:

1. The smell "No Dynamic Page" for entity type Customer is eliminated by the application of refactoring "Insert Dynamic Pages": Two dynamic pages (an index page and a data page) referencing entity type Customer are inserted into the hypertext layer. Moreover, the inserted data page is linked by the index page which is in turn linked by the starting page.
2. The smell "Missing Data Page" for index page agencyindex is eliminated by the application of refactoring "Add Data Page to Index Page": A data page

which shows Agency entities and which is linked by the index page agencyindex is inserted.
3. In order to eliminate the smell "Empty Entity", a new reference address is inserted for entity type Agency.
4. Finally, some missing information is supplemented: A new entity type BankAccount which is referenced by entity type Customer is added to the data layer. Moreover, an attribute postalcode is inserted for entity type Address, attributes manufacturer and power are inserted for entity type Car.

The result of comparing the initial model v_1 and its improved revision v_2 using the Eclipse built-in textual diff utility is shown in Fig. 1. Similar results are obtained using other graphical difference tools such as Meld [24] or KDiff3 [15]. The textual output produced by the UNIX `diff` utility [23] reports 8 deletions and 23 insertions of lines of text. These examples illustrate that the line-based approach fails to explain the improvements on our sample SWML model in an adequate way.

3 High-level Differencing of Textual Models

In this section, we briefly review our approach to high-level model differencing. Next, we describe how to extend the approach and tooling to textual models and finally present our reference implementation which is based on standard Eclipse Modeling technologies and fully integrated with Xtext.

3.1 Approach

In [18], we introduce an approach to high-level differencing which works on a structural representation of two model versions v_1 and v_2 which are to be compared. A model is conceptually regarded as typed, attributed, directed graph which is known as the abstract syntax graph (ASG) of this model. The difference calculation basically proceeds in three steps:

1. Initially, a matching procedure identifies corresponding nodes and edges which are considered to be "the same" elements in v_1 and v_2.
2. Subsequently, a low-level difference is derived. Elements not involved in a correspondence are considered to be deleted or created, each non-identical attribute value of corresponding elements is considered to be updated.
3. Finally, an operation detection algorithm recognizes executions of edit operations in the low-level difference. The available edit operations are provided as additional input parameter, each operation has to be formally specified as a transformation rule in the model transformation language Henshin [4].

Similar to the UNIX `diff` utility, a calculated difference $\Delta(v_1, v_2)$ is a description of how model v_1 can be edited to become revision v_2 in a step-wise manner. However, the available edit operations are defined on the ASG which enables us to report edit steps on a much higher level of abstraction. In principle, any language-specific restructuring operation can be supported as long as it can be specified in a Henshin transformation rule. In other words, we consider the effect of an edit step as an in-place model transformation which is formally specified as a declarative transformation rule to which we refer as edit rule. Thus, the set of available edit operations can be specifically tailored for a given modeling language. An example edit rule for SWML models is briefly explained in Appendix B.

Since the approach presented in [18] has been developed in the context of visual modeling languages, it assumes the allowed types of nodes and edges of an ASG to be defined by a meta-model. Nonetheless, although our approach typically starts with a meta-model, it can be applied to textual DSMLs, too. We only require a procedure which converts the grammar into a meta-model, e.g. as presented in [2,28,6].

3.2 Tool Architecture

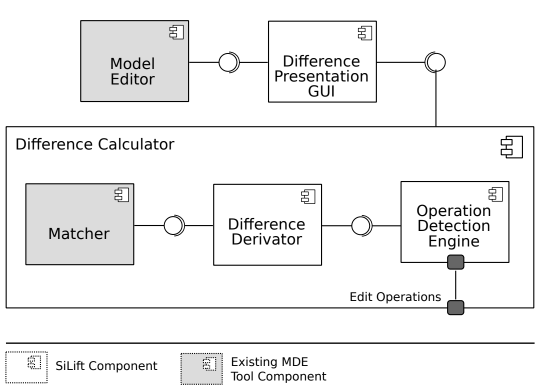
An overview of the core components of a difference tool which implements our approach is shown in Fig. 2a. Exchangeable components which are typically provided by an existing MDE environment are colored in light gray.

The Difference Calculator calculates a difference in a step-wise manner according to our conceptual approach. Consequently, the sub-components Matcher, Difference Derivator and Operation Detection Engine are arranged in a pipeline. A calculated difference is presented to developers in an interactive Difference Presentation GUI as shown in Fig. 3. A control window on the left lists the edit steps. The effect of an edit step is explained on the basis of the concrete syntax. To that end, the original and changed model are displayed in their standard editor on the right. Selecting an edit step in the control window causes the context of this edit step to be highlighted in the respective editor windows. In principle, the GUI can be integrated with any model editor. We only require that the editor offers an API such that external representation of a model element, i.e. certain characters, lines of text or text blocks, can be highlighted.

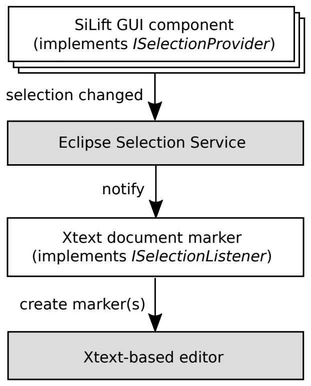
3.3 Integration with the Xtext Language Development Framework

An EMF-based reference implementation of our approach is available within the model versioning framework SiLift [27]. In this work, we extend the SiLift framework by an integration with the widely used language development environment Xtext [30]. A download option is provided at the accompanying web site of this paper [1].

The adaptation of the algorithmic components is straightforward since an Ecore-based meta-model can be automatically generated by Xtext. The integration of the difference presentation GUI is illustrated in Fig. 2b. The GUI is loosely coupled with generated Xtext editors via the Eclipse Selection Service. All SiLift sub-windows implement the ISelectionProvider interface and thus report which conceptual model elements are currently selected. The selection service notifies registered ISelectionListeners about selection changes induced by a selection



(a) Core difference tool components


SiLift GUI component (implements *ISelectionProvider*)
selection changed
Eclipse Selection Service
notify
Xtext document marker (implements *ISelectionListener*)
create marker(s)
Xtext-based editor

(b) Integration with Xtext

Fig. 2: Tool architecture and integration with the Xtext framework

provider. Our selection listener implementation is based on the Eclipse marker framework which can be used to highlight text fragments in textual Eclipse editors. In order to get the position of a conceptual model element within the textual representation of a model, we utilize the Xtext contribution to the EMF adapter mechanism: For each EObject which originates from an Xtext resource, we get an adapter for this EObject providing access to the corresponding node of the Xtext parse tree. The nodes in a parse tree provide the required position information.

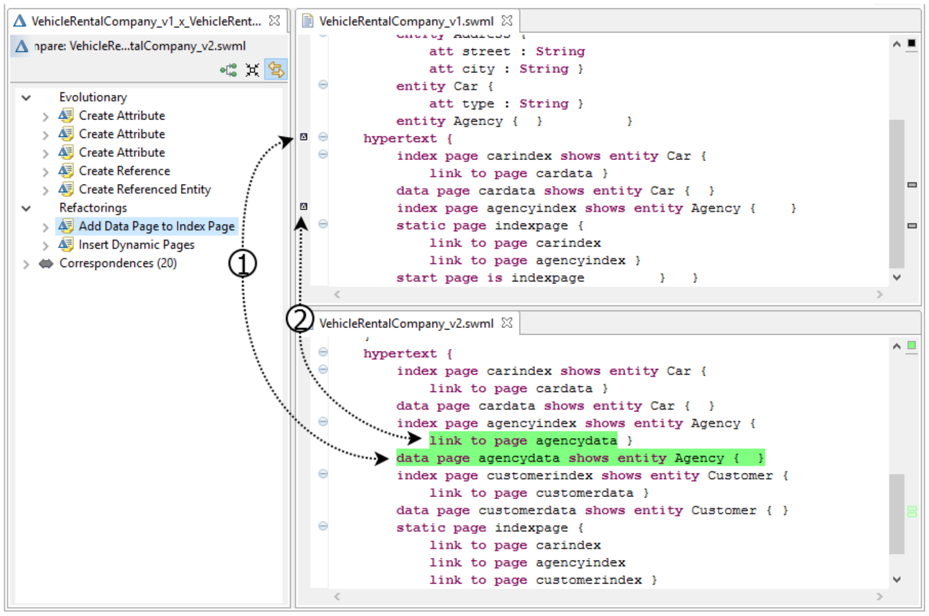
4 Adaptation and Application to SWML

In this section, we outline the adaptation of our tool environment to the SWML. An overview of the difference calculation configuration is given in Sec. 4.1. Finally, Sec. 4.2 presents the results of applying our difference tool which uses this configuration to the example change scenario of Sec. 2.2.

4.1 Configuration of the Difference Calculation

Two of the core differencing components of Fig. 2a have to be adapted to SWML, the matcher and the operation detection engine.

To determine corresponding elements in SWML models, we implemented a signature-based matching strategy [17] using the Epsilon Comparison Language (ECL) [20]. ECL is a domain-specific language for developing highly customized model comparison rules, our SWML matching configuration can be found in the Appendix A. Singleton objects of types WebModel, DataLayer, HypertextLayer are matched immediately. Names of named model elements (Entity, Attribute, etc.) are used as unique signature values, i.e. correspondences are established between


VehicleRentalCompany_v1_x_VehicleRent...
npare: VehicleRe...talCompany_v2.swml
Evolutionary
Create Attribute
Create Attribute
Create Attribute
Create Reference
Create Referenced Entity
Refactorings
Add Data Page to Index Page
Insert Dynamic Pages
Correspondences (20)
1
2
VehicleRentalCompany_v1.swml
VehicleRentalCompany_v2.swml

```
            att street : String
            att city : String }
        entity Car {
            att type : String }
        entity Agency {  }           }
    hypertext {
        index page carindex shows entity Car {
            link to page cardata }
        data page cardata shows entity Car {  }
        index page agencyindex shows entity Agency {     }
        static page indexpage {
            link to page carindex
            link to page agencyindex }
        start page is indexpage          }   }
```

```
    hypertext {
        index page carindex shows entity Car {
            link to page cardata }
        data page cardata shows entity Car {  }
        index page agencyindex shows entity Agency {
            link to page agencydata }
        data page agencydata shows entity Agency {  }
        index page customerindex shows entity Customer {
            link to page customerdata }
        data page customerdata shows entity Customer { }
        static page indexpage {
            link to page carindex
            link to page agencyindex
            link to page customerindex }
```

Fig. 3: SiLift SWML application: Difference between sample models v_1 and v_2

equally named elements having the same type. Finally, Link objects representing hyperlinks between web pages are matched if they connect the same pages, i.e. if the source and target pages of two links are matched.

The specification of edit rules is supported by one of our meta-tools known as SiDiff Edit Rule Generator (SERGe) [26]. SERGe derives sets of basic edit rules from a given meta-model with multiplicity constraints. These sets are complete in the sense that all kinds of edit rules, i.e. create, delete, move and change operations, are contained for every node type, edge type and attribute defined by the meta-model. For SWML, 29 basic edit rules have been generated. In addition, we manually specified 9 complex edit rules, 5 refactorings which could be re-used from the EMF Refactor tool environment [11], and 4 evolutionary edit operations which facilitate frequently recurring editing tasks. As an example, the edit rule for refactoring "Add Data Page to Index Page", which is applied in step 2 of our motivating example of Sec. 2.2, is shown in Appendix B. The complete set of edit rules is included in the SWML configuration, which is available from the Eclipse update site at [1].

4.2 Application to the Example Change Scenario

Fig. 3 presents the results of applying our difference tool that uses the above configuration to our example of Sec. 2.2. The edit step "Add Data Page to Index Page" is currently selected and its effect can be inspected more closely in the editor windows on the right. The inserted data page agencydata (s. marker indicated by ① in Fig. 3) and the new link to this page from the index page agencyindex (s. marker ② in Fig. 3) are easy to see in the lower editor window which shows version v_2. In the upper editor window showing version v_1, we can see the context of the respective changes, e.g., data page agencydata has been inserted in the hypertext layer. In a similar way, one can interactively inspect the other refactoring "Insert Dynamic Pages" (s. change 1 in Sec. 2.2) and the evolutionary edit steps representing changes 3 and 4 of the change scenario of Sec. 2.2.

5 Related Work

In this section, we briefly review related work regarding the two main aspects of model difference tools addressed in this paper, namely *i)* the adaptability to a new language, and *ii)* the integration with an MDE environment, thereby putting a special emphasis on EMF technologies.

Many approaches and tools to model differencing have been proposed recently, surveys can be found in [13,3]. Similar to ours, virtually all of them work on a structural representation of models. However, only a few of them are adaptable to a new modeling language and almost all of them use primitive graph operations such as creating/deleting single nodes/edges as edit operations for ASGs. The recognition of complex changes such as language-specific refactorings seems to be supported only by few approaches, e.g. [21,22,29]. A detailed review of how these approaches differ from ours can be found in [18,19]. To the best of our knowledge, none of them has yet been adapted to textual DSMLs, which is the main contribution of the tool environment presented in this paper.

A dedicated difference presentation GUI is offered by only a few EMF-based difference tools for models. EMF Compare [9], the currently most widely used differencing tool for EMF-based models, displays two versions of a model in parallel in their abstract syntax tree representation. A similar approach is implemented in EMF Diff/Merge [10] and the RSA tool suite [14]. The parallel display largely fails to present complex model changes. Again, to the best of our knowledge, none of the existing EMF tools can be used with Xtext editors in an integrated way.

6 Conclusion and Future Work

In this paper, we presented concepts and a tool environment to flexibly specify and recognize complex changes in textual models. The tooling, called SiLift, is based on EMF and tightly integrated with the widely used Xtext framework. It enables developers to understand complex structural changes in textual models and is an attractive alternative to traditional line-based difference tools. Moreover, the obtained differences can be converted to executable edit scripts [19] serving as a basis for model patching and structural merging [25].

Obviously, the proposed solutions become more powerful from a practical point of view if they are tightly integrated into an existing version control system such as Git or Subversion. We leave such an integration for future work.

Acknowledgments

This work was partially supported by the DFG (German Research Foundation) under the Priority Programme SPP1593: Design For Future - Managed Software Evolution. Moreover, the authors would like to thank Thorsten Arendt for providing us the quality assurance scenario of the SWML case study.

References

1. Accompanying materials for this paper: http://pi.informatik.uni-siegen.de/projects/SiLift/ocl2015.php; 2015
2. Alanen, M.; Porres, I.: A relation between context-free grammars and meta object facility metamodels; Technical Report No. 606, Turku Centre for Computer Science; 2004
3. Altmanninger, K.; Seidl, M.; Wimmer, M.: A Survey On Model Versioning Approaches; p. 271-304 in: Intl. Journal of Web Information Systems 5:3; 2009
4. Arendt, T.; Biermann, E.; Jurack, S.; Krause, C.; Taentzer, G.: Henshin: Advanced Concepts and Tools for In-Place EMF Model Transformations; in: Proc. MoDELS 2010, LNCS 6394, Springer; 2010
5. Arendt, T.; Taentzer, G.; Weber, A.: Quality Assurance of Textual Models within Eclipse using OCL and Model Transformations; in: Proc. OCL @ MoDELS; 2013
6. Bergmayr, A.; Wimmer, M.: Generating Metamodels from Grammars by Chaining Translational and By-Example Techniques; in: Proc. MDEBE @ MoDELS; 2013
7. Brambilla, M.; Cabot, J.; Wimmer, M.: Model-driven software engineering in practice; p. 1-182 in: Synthesis Lectures on Software Engineering; 1(1); 2012
8. EMF: Eclipse Modeling Framework; http://www.eclipse.org/emf; 2015
9. EMF Compare; http://www.eclipse.org/emf/compare; 2015
10. EMF Diff/Merge; http://eclipse.org/diffmerge; 2015
11. EMF Refactor; https://www.eclipse.org/emf-refactor; 2015
12. Estublier, J.; Leblang, D.; van der Hoek, A.; Conradi, R.; Clemm, G.; Tichy, W.; Wiborg-Weber, D.: Impact of software engineering research on the practice of software configuration management; p. in 383-430: ACM Trans. on Software Engineering and Methodology 14:4; 2005
13. Förtsch, S.; Westfechtel, B.: Differencing and Merging of Software Diagrams - State of the Art and Challenges; p. 90-99 in: Proc. Int. Conf. Software and Data Technologies; 2007
14. IBM Rational Software Architect; http://www.ibm.com/developerworks/rational/products/rsa; 2015
15. KDiff3: http://kdiff3.sourceforge.net; 2014
16. Kehrer, T.; Kelter, U.; Ohrndorf, M.; Sollbach, T.: Understanding Model Evolution through Semantically Lifting Model Differences with SiLift; p. 638-641 in: Proc. 28th IEEE Int. Conf. on Software Maintenance; 2012
17. Kehrer, T.; Kelter, U.; Pietsch, P., Schmidt, M.: Adaptability of Model Comparison Tools; in: Proc. 27th Int. Conf. on Automated Software Engineering; 2012

18. Kehrer, T.; Kelter, U.; Taentzer, G.: A Rule-Based Approach to the Semantic Lifting of Model Differences in the Context of Model Versioning; p. 163-172 in: Proc. 26th Int. Conf. on Automated Software Engineering; 2011
19. Kehrer, T.; Kelter, U.; Taentzer, G.: Consistency-Preserving Edit Scripts in Model Versioning; p.191-201 in: Proc. 28th Int. Conf. on Automated Software Engineering; 2013
20. Kolovos, D.: Establishing Correspondences between Models with the Epsilon Comparison Language; p. 146-157 in: Proc. Intl. Conf. on Model Driven Architecture-Foundations and Applications; 2009
21. Könemann, P.: Capturing the Intention of Model Changes; p.108-122 in: Proc. Int. Conf. on Model Driven Engineering Languages and Systems; 2010
22. Langer, P.; Wimmer, M.; Brosch, P.; Herrmannsdörfer, M.; Seidl, M.; Wieland, K.; Kappel, G.: A posteriori operation detection in evolving software models; p. 551-566 in: Journal of Systems and Software 86(2); 2013
23. MacKenzie, D.; Eggert, P.; Stallman, R.: Comparing and Merging Files with GNU diff and patch; Network Theory Ltd.; 2003
24. Meld: http://meldmerge.org; 2015
25. Mens, T.: A State-of-the-Art Survey on Software Merging; p. 449-462 in: IEEE Trans. Software Eng. 28:5; 2002
26. Rindt, M.; Kehrer, T.; Kelter, U.: Automatic Generation of Consistency-Preserving Edit Operations for MDE Tools; in: Proc. Demonstrations Track of MoDELS; 2014
27. SiLift project; Semantic Lifting of Model Differences; http://pi.informatik.uni-siegen.de/projects/SiLift; 2015
28. Wimmer, M.; Kramler, G.: Bridging grammarware and modelware; in: Satellite Events at the MoDELS 2005 Conference; 2005
29. Xing, Z.; Stroulia, E.: Refactoring detection based on UMLDiff change-facts queries; p.263-274 in: Working Conf. on Reverse Engineering; 2006
30. Xtext; http://eclipse.org/Xtext; 2015

A SWML Matching Configuration Implemented in ECL

```
rule WebModel2WebModel
      match left : Left!WebModel
      with right : Right!WebModel {
      compare {
            return true;
      }
}
// Same for DataLayer and HypertextLayer
// ...

rule Entity2Entity
      match left : Left!Entity
      with right : Right!Entity {
      compare {
            return left.name = right.name;
      }
}
```

```
// Same for Attribute, Reference and Page
// ...

rule Link2Link
    match left : Left!Link
    with right : Right!Link {
    compare {
        return left.srcMatches(right) and left.tgtMatches(right);
    }
}
operation Link srcMatches(other : Link) : Boolean {
    return self.eContainer.name = other.eContainer.name;
}
operation Link tgtMatches(other : Link) : Boolean {
    return self.target.name = other.target.name;
}
```

Listing A-1: SWML Matching Configuration Implemented in ECL

B Refactoring ”Add Data Page to Index Page“ Implemented in Henshin

Fig. 4 shows how to implement the refactoring operation ”Add Data Page to Index Page“ in Henshin. The example illustrates that Henshin offers an intuitive visual syntax to specify model patterns to be found and preserved, to be deleted and to be created. Note that selectedEObject and entityname are input parameters, while New_DataPage and New_Link are output parameters of the rule. The change actions which are to be performed by the rule are specified based on the SWML abstract syntax. Thus, the specification uses type definitions of the SWML meta-model which is generated by the Xtext framework. Given an index page selectedEObject which references an entity named entityname, a new data page New_DataPage referencing this entity is created. Moreover, a new link New_Link is created such that the inserted data page is linked by the index page.

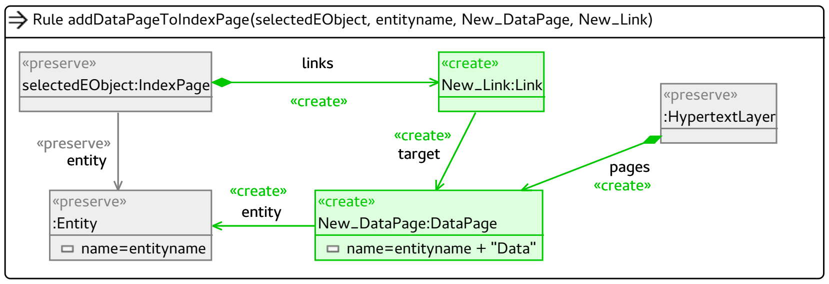


Fig. 4: Refactoring ”Add Data Page to Index Page“ implemented in Henshin