Henshin: Advanced Concepts and Tools for In-Place EMF Model Transformations

Thorsten Arendt[1], Enrico Biermann[2], Stefan Jurack[1], Christian Krause[3,⋆], Gabriele Taentzer[1]

[1] Philipps-Universität Marburg, Germany
{arendt,sjurack,taentzer}@mathematik.uni-marburg.de
[2] Technische Universität Berlin, Germany
enrico@cs.tu-berlin.de
[3] CWI Amsterdam, The Netherlands
c.krause@cwi.nl

Abstract. The Eclipse Modeling Framework (EMF) provides modeling and code generation facilities for Java applications based on structured data models. Henshin is a new language and associated tool set for in-place transformations of EMF models. The Henshin transformation language uses pattern-based rules on the lowest level, which can be structured into nested transformation units with well-defined operational semantics. So-called amalgamation units are a special type of transformation units that provide a forall-operator for pattern replacement. For all of these concepts, Henshin offers a visual syntax, sophisticated editing functionalities, execution and analysis tools. The Henshin transformation language has its roots in attributed graph transformations, which offer a formal foundation for validation of EMF model transformations. The transformation concepts are demonstrated using two case studies: EMF model refactoring and meta-model evolution.

1 Introduction

Model-driven software development (MDD) is considered as a promising paradigm in software engineering. Models are ideal means for abstraction and enable developers to master the increasing complexity of software systems.

In model-driven development, the transformation of models belongs to the essential activities. Since models become the central artifacts in MDD, they are subject to direct model modifications, translated to intermediate models, and finally code is generated. While direct model modifications are usually performed in-place, i.e. directly on the model without creating copies, model translations usually keep source models untouched and produce new models or code. These transformations are called out-place.

Another crucial concept for MDD are domain-specific modeling languages which allow the definition of models on an adequate abstraction level with all

⋆ Supported by the NWO GLANCE project WoMaLaPaDiA.

D.C. Petriu, N. Rouquette, Ø. Haugen (Eds.): MODELS 2010, Part I, LNCS 6394, pp. 121–135, 2010.
© Springer-Verlag Berlin Heidelberg 2010

information needed to generate the right models or code. A promising approach to define domain-specific modeling languages is the Eclipse Modeling Framework (EMF) [1,2] which has evolved to a well-known and widely used technology. EMF provides modeling and code generation capabilities based on so-called structural data models. As they describe structural aspects only, they are mainly used to specify domain-specific languages. EMF complies with Essential MOF (EMOF) as part of OMG's Meta Object Facility (MOF) 2.0 specification [3].

For various kinds of EMF model modifications such as refactorings, introduction of design patterns and other modeling patterns, we need a powerful in-place transformation approach, operating directly on EMF models. There are several in-place model transformations approaches which can transform EMF models directly, e.g. Kermeta [4], EWL [5], EMF Tiger [6], and Moment2 [7]. The corresponding transformation languages are either rather simple or in case the of Kermeta, not declarative enough to offer the opportunity for formal reasoning on model transformations.

To fill this gap, we have developed the transformation language and tool environment Henshin, operating directly on EMF models. Henshin is a successor of EMF Tiger in the sense that it is also based on graph transformation concepts but extends the transformation language of EMF Tiger considerably. Henshin comes along with a powerful, yet declarative model transformation language, offering the possibility for formal reasoning. Its basic concept of transformation rules is enriched by powerful application conditions and flexible attribute computations based on Java or JavaScript. Furthermore, it provides the concept of transformation units defining control structures for rule applications in a modular way. A special kind of transformation unit are amalgamation units which offer a forall-operator for applying transformation rules in parallel. For further flexibility, special units for code execution can be added.

The Henshin tool environment consists primarily of a fast transformation engine, several editors, and a state space generator to support reasoning by model checking based on state space generation from transformation systems, useful for model checking transformations. Since these transformation concepts are close to graph transformation concepts, it is possible to translate the rules to AGG [8], a tool environment for algebraic graph transformation where they might be further analyzed concerning conflicts and dependencies of rule applications as well as their termination.

Two example applications of Henshin are considered: (1) refactoring of EMF models [9], more precisely refactoring *Pull Up Attribute* and (2) a simple form of meta-model evolution [10] where two evolution steps of a Petri net model are reflected on instance models being concrete Petri nets in abstract syntax.

The paper is organized as follows: Section 2 introduces the Henshin transformation language and describes its most important concepts. In Section 3 and 4, we present two case studies on refactoring and meta-model evolution. The Henshin tool environment is presented in Section 5. A discussion of related work can be found in Section 6 and concluding remarks in Section 7.

2 The Henshin Transformation Meta-model

In the following, we describe informally our transformation language using the Henshin transformation meta-model, which is also an EMF meta-model and moreover uses the Ecore meta-model for typing purposes. The Henshin transformation language is based on graph transformation concepts [11,12,13] and therefore offers a visual syntax and means for formal reasoning about transformations.

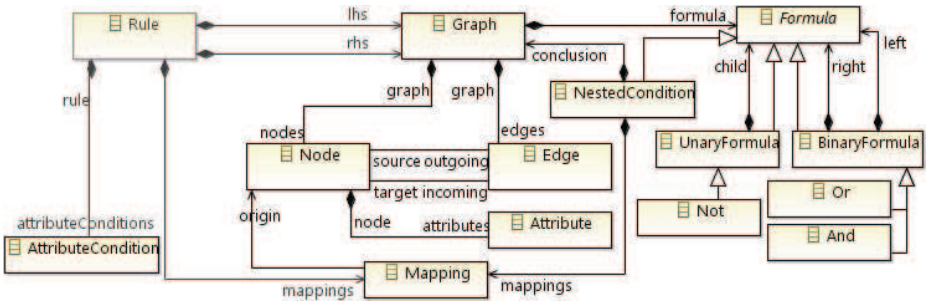

Fig. 1. Rules with application conditions

2.1 Rules and Matching

A transformation rule consists of left and right-hand side graphs (respectively LHS and RHS) which describe model patterns by their underlying (graph) structure (cf. Fig. 1). Furthermore, attribute conditions can be defined for rules. Nodes refer to objects while edges refer to references between objects. Nodes, edges and attributes refer to *EClass*, *EReference*, and *EAttribute* via references called *type* (not shown in Fig. 1) which are classes of the Ecore meta-model. These type references are used as an explicit typing, e.g. a node connected to a certain *EClass* will will match only to objects of this type. *Mappings* between LHS and RHS can be defined between nodes. Since EMF models cannot contain parallel edges of the same type between the same nodes, edge mappings are implicitly given if both, their source and target nodes are mapped. For clarity, we omit the explicit notation of multiplicities in all figures. As a general guideline, all reference types using names in plural have 0..* multiplicity. All others have upper bound 1. Note that we concentrate on the structure of the transformation meta-model and neglect the properties of model elements such as their names.

Rules can be applied to a construct called *EmfGraph* that serves as an aggregation of *EObjects*. Only the *EObjects* within an *EmfGraph* will be considered for matching. Therefore, deleting *EObjects* removes them from the underlying *EmfGraph* representation only. The EObject might still be used in another context but it is no longer visible for further rule applications.

2.2 Application Conditions

To conveniently determine where a specified rule should be applied, application conditions can be defined. An important subset of application conditions are negative application conditions (NACs) which specify the non-existence of model patterns in certain contexts.

Application conditions allow the definition of first order logical formulas over graph conditions, being atomic conditions that enforce the existence or non-existence of model patterns, as well as further conditions over conditions (nesting). Statements like "a node must have an incoming edge or an outgoing edge" or "a node that is connected to this node may not have a looping edge" can be easily expressed.

In the Henshin transformation model, shown in Fig. 1, *Graphs* can be annotated with application conditions using a *Formula*. This formula is either a logical expression or an application condition which is an extension of the original graph structure by additional nodes and edges. A rule can be applied to a host graph only if all application conditions are fullfilled.

2.3 Transformation Units

To control the order of rule applications, it is possible to define control structures over rules called *TransformationUnits*. The most basic transformation unit is a rule itself which corresponds to a single application of that rule. All available transformation units are depicted in Fig. 2. For example, there are constructs for non-deterministic rule choices (*IndependentUnit*) and rule priority (*PriorityUnit*). Except for *Rules* and *AmalgamationUnits*, transformation units can have one or more subunits which are executed according to the semantics of its parent unit. For instance, subunits of an *IndependentUnit* will be executed in random order.

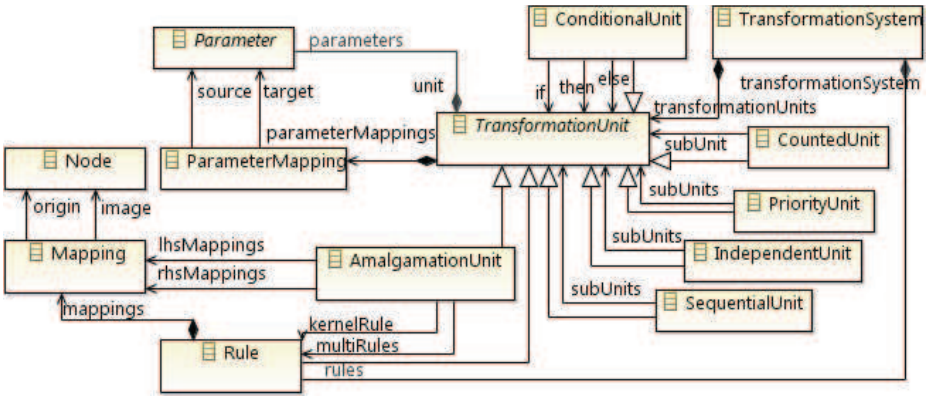

Fig. 2. Transformation units and parameters

Furthermore, it is possible to pass objects and values from one unit to another one via parameters. In this way, the object flow between different rules and units can be controlled and complex transformations can be parameterized. Each transformation unit can have an arbitrary number of *Parameters* which can either refer to a specific EObject or contain a specific value. *ParameterMappings* define how parameters of transformation units are passed to their subunits.

Applicability. A unit is applicable if it can be successfully executed. Applicability is defined differently for different transformation units. For example, *PriorityUnit*s or *IndependentUnit*s are always applicable while a *SequentialUnit* is applicable only if all of its subunits are applicable in the given order.

Termination. A unit terminates if it is successfully executed or if no rule was applied in the context of that unit. *ConditionalUnit*s or *SequentialUnit*s terminate if their subunits terminate. However, subunits of *PriorityUnit*s and *IndependentUnit*s may be applied repeatedly. This can easily result in infinite loops when nesting units of those kinds. *IndependentUnit*s and *PriorityUnit*s terminate if their subunits do not contain any applicable rule.

2.4 Amalgamation

A special kind of transformation units are *AmalgamationUnit*s which are useful to specify forall operations on recurring model patterns. An amalgamation unit contains an interaction scheme consisting of one *Rule* which acts as a kernel rule and multiple rules which act as multi rules. The embedding of a kernel rule into a multi rule are defined by *Mappings* between nodes of the LHS of the kernel and the multi rule. The semantics of such an interaction scheme is that the kernel rule is matched exactly once. This match is used as a common partial match for each multi rule which are matched as often as possible. The effect is that the modification defined in the kernel rule is applied only once while modifications defined in the multi rules are applied a certain number of times depending on the number of matches. For a detailed presentation of amalgamation concepts, see [12]. An amalgamation unit is applicable if its kernel rule is applicable. It terminates after one application.

2.5 Relation to Algebraic Graph Transformation

The presented language concepts of Henshin have their origin in algebraic graph transformation [11]. This concerns the syntactical structure of rules and transformation units as well as their semantics wrt. *EMFGraphs*. While an *EMFGraph* corresponds to a typed, attributed graphs, the given EMF model represents the type graph. Nodes and edges in rules are related to EObjects and EReferences which are typed over the same given EMF model. Formulas relate to graph conditions [11] over typed, attributed graphs. The amalgamation concept is formulated for typed graphs with node type inheritance and containment in [12]. Finally, transformation units are defined in [13] using an approach-independent

form. However, parameters have not been considered yet in the formal setting, but will be in future work.

To summarize, our general aim is to give a formal semantics to the full transformation language as solid basis for further validations. A large foundation is already available and will be completed in the near future.

3 EMF Model Refactoring

In this section we present an example refactoring for EMF based models [1] using the advanced concepts of Henshin.

3.1 DSL *SimplifiedClassModel (SCM)*

Figure 3 shows the meta-model of DSL *SimplifiedClassModel (SCM)* for modeling simplified class diagrams being useful in an early stage of the software development process to formulate analysis models. SCM can be considered as simplification of the UML superstructure [14]. Meta-attributes and references *name*, *qualifiedName*, *visibility*, and *redefinedAttribute*, as well as well-formedness rules correspond to those known from UML and are not explained in detail here.

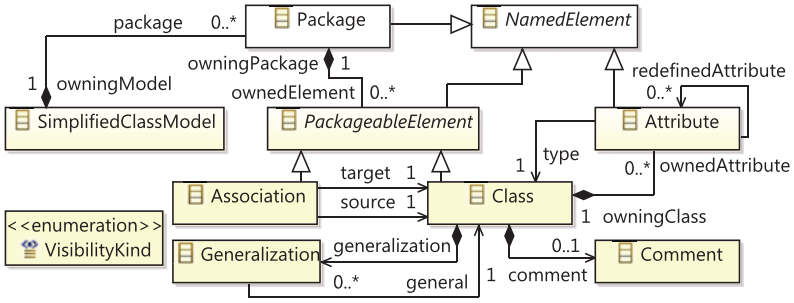

Fig. 3. DSL *SimplifiedClassModel (SCM)* - meta-model

3.2 Model Refactoring *Pull Up Attribute*

SCM refactoring *Pull Up Attribute* moves a common attribute from all direct subclasses of a given class to this class. The name of the attribute to be moved is given by parameter `attributename` while parameter `superclassname` specifies the qualified name of the class the attribute has to be pulled up to. In order to apply *Pull Up Attribute*, the following preconditions (**PC**) have to be checked:

– The class with qualified name `superclassname` does not already have an attribute named `attributename` (**PC1**).

– For each direct subclass of the class with qualified name `superclassname`:
  - There is an attribute named `attributename` (**PC2**).
  - Visibility (**PC3**) and type (**PC4**) of the attribute named `attributename` are the same.
  - If the attribute named `attributename` redefines another attribute each attribute named `attributename` in each other subclass of the class with qualified name `superclassname` has to redefine the same attribute (**PC5**). Furthermore, the redefined attribute must have visibility *private*, i.e. it must not be visible in class with qualified name `superclassname` (**PC6**).

If each precondition is fulfilled, the class with qualified name `superclassname` gets a new attribute named `attributename`. Corresponding attributes are removed from all subclasses. The new attribute gets the same visibility as before, except that visibility *private* has to be set to *protected* since a subclass must have access to the new attribute as well. Moreover, already redefined attributes have to be referenced by the new attribute.

3.3 Implementation Using Henshin

The Henshin implementation of *Pull Up Attribute* uses a *SequentialUnit* which in turn uses three *IndependentUnits* as subunits. The first *IndependentUnit* is responsible for preconditions checking and contains six rules. Each rule is specified in a way that the class with qualified name `superclassname` gets a comment 'ERROR' if a certain precondition is violated.

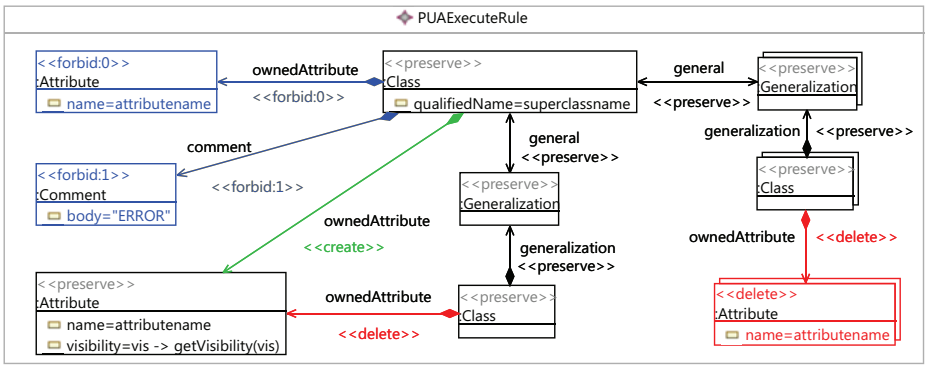

Fig. 4. Rule *PullUpAttributeRule*

The second *IndependentUnit* performs the transformation using an *AmalgamationUnit*. Figure 4 shows two rules (the *kernel* and a *multi rule*) of the *AmalgamationUnit* as well as their LHS, RHS, and two NACs in an integrated view. LHS objects (nodes and edges) can be identified by tags ⟪*preserve*⟫ or ⟪*delete*⟫, objects tagged by ⟪*preserve*⟫ or ⟪*create*⟫ form the RHS of the rule. Kernel rule

nodes are bordered by a single line, whereas the multi rule contains all those of the kernel rule and those objects bordered by two lines. They represent so-called multi-objects.

The kernel rule moves the attribute from a class to its superclass. In its LHS we are looking for an attribute named `attributename` contained in a subclass of the class with qualified name `superclassname`. A condition on attribute *visibility* changes its value to *protected* only if its previous value was *private*. This is done by invoking Java method `getVisibility()` where the previous visibility is given by variable *vis*. There are two NACs which have to be checked before executing the specified transformation ($\langle\langle forbid : 0 \rangle\rangle$ and $\langle\langle forbid : 1 \rangle\rangle$). The first NAC checks whether the superclass has not been annotated by comment 'ERROR', whereas the second one checks whether the superclass does not already own an attribute named `attributename`. After rule application the class with qualified name `superclassname` owns the attribute named `attributename`.

The multi rule deletes the corresponding attribute from each further subclass. Its LHS corresponds to the kernel rule LHS enriched by a sub-pattern for possibly other subclasses of the class with qualified name `superclassname` that own an attribute named `attributename`. This additional pattern is matched into the model graph as often as different further subclasses exist. According to the $\langle\langle delete \rangle\rangle$ tagged multi-object the corresponding attribute will be removed from each further subclass.

The third *IndependentUnit* consists of one single rule that removes comment 'ERROR' possibly inserted before. Each part of the refactoring (checking, performing, and cleaning) has to be encapsulated by an *IndependentUnit* in order to assure a successful execution of *Pull Up Attribute*, i.e. the target model is valid either if the refactoring has been actually performed or not because of violated conditions. Please note that the complete specification of *Pull Up Attribute* can be found at [15].

4 Towards Meta-model Evolution

In model-driven and model-based development models are the key artifacts. As it is quite natural that models evolve over time the compliance of existing instances with such meta-models needs to be obtained. Not all modifications of a meta-model lead to invalidity. In [10], Cicchetti et al. propose three categories of model changes: *Not breaking changes* occur without breaking model instances, *breaking and resolvable changes* break the instances but can be resolved by automatic means and furthermore, *breaking and unresolvable changes* are those which do break the instances and which cannot be resolved automatically.

In our case study below, we follow the *manual specification* approach, i.e. we encode meta-model and instance model changes manually since currently there does not exist a meta-model evolution framework based on Henshin. Nevertheless, we give a practical idea how (semi-) automatic meta-model evolution can be realized with Henshin leading to an *operator-based co-evolution* approach.

Henshin is able to handle any Ecore-based model, thus we can create transformation rules for both, meta-models and its instances. In general, meta-models

may occur in form of an Eclipse plug-in with generated model classes or standalone as *.ecore* file. The latter is more flexible and since Henshin supports Dynamic EMF, we use such Ecore files in our approach. In the following case study, the control flow is currently implemented in form of a simple Java class which loads related models and transformation rules and which triggers the transformation performed by the Henshin interpreter. This implementation as well as corresponding models and rules are part of our Henshin examples plug-in [16].

Our case study is dealing with the evolution of a Petri net meta-model. Figure 5 shows a simple Petri net meta-model on the left while an evolved one is shown on the right. A simple Petri net contains `Place`s and `Transition`s which can be interconnected by dedicated references. `Net` serves as root node. The enhanced meta-model provides further nodes, `ArcPT` and `ArcTP`, serving as connection entries between `Place`s and `Transition`s or `Transition`s and `Place`s, respectively. Since `ArcPT` and `ArcTP` inherit from abstract `Arc`, it can be used to specify a weight. Complying Petri net instances can be deduced easily and are not shown due to space constraints.

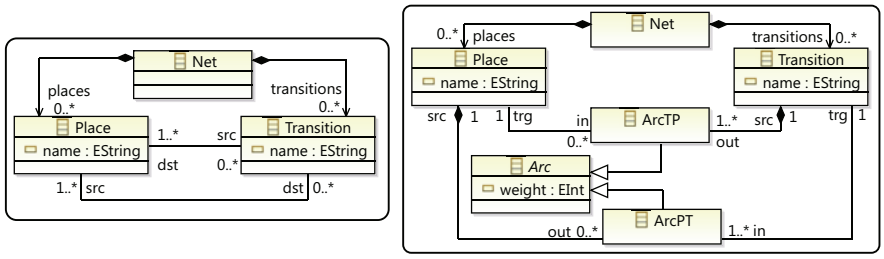

Fig. 5. Evolving Petri net meta-models. The original model is shown on the left while the evolved model is shown on the right.

In order to perform an evolution as shown in Fig. 5, the utilization of a set of general rules is conceivable being applied in a certain order. For example, the first iteration step may be a replacement of a connection between two classes by a connection class. The next iteration step may be to *extract a super-class* analog to the well-known corresponding refactoring. Afterwards the attribute could be introduced into the super-class. Each meta-model modification comes with an adaption of its instance models. In the following we concentrate on the replacement of connections only to demonstrate meta-model evolution with Henshin. Such a replacement rule may be modeled in a very general way as shown in Fig. 6. While two classes with given names and their container package are *preserve*d, two references shown in the upper area are *delete*d and another four references and one class are *create*d. Two *eOpposite* references between each two *EReference*s are omitted to keep the rule compact here. A negative application condition checks for the existence of a class named equally to the introduced class since doublets are *forbid*den. Note that *srcName*, *trgName* and

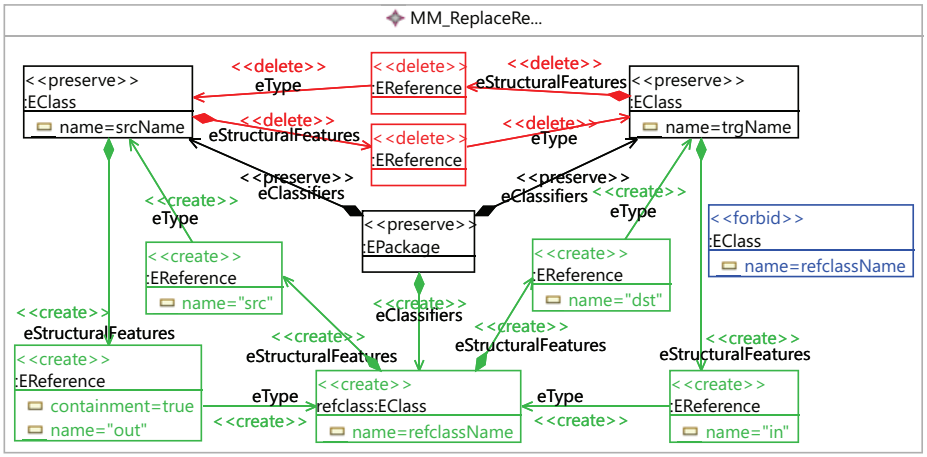

Fig. 6. General rule for replacing a connection by a connection class

refclassName are so-called *parameters* representing the names of the connected classes and the new reference class. They have to be set before the application of the rule. In order to maintain compliance of instance models, e.g. meta-model elements in use must not be removed, our evolution step of replacing connections is structured in several sub-steps as follows. Note, in our case these steps could be deduced even automatically.

The first step is to create new types and references. The creation part of Fig. 6 leads to such a rule. The deletion part has to occur in that rule as well but in terms of a preserved part. Having match and co-match, this allows to maintain information about concrete classes and references to be replaced. For the following we assume the parameters are set as follows: *srcName*="Place", *trgName*="Transition" and *refclassName*="ArcPT". The second step is to modify all instance models such that old direct references are deleted and replaced by instances of the new class, each referred to by an instance of the source and target class. With the previous rule, its match and its co-match at hand, the generation of a rule as depicted in Fig. 7 targeting instance model changes is quite conceivable. The rule may be embedded into an independent unit additionally. In this case it is applied as often as possible, i.e. the replacement takes place sequentially. A parallel replacement would be possible as well by utilizing an amalgamation unit. In that case the kernel rule would be empty and the rule in Fig. 7 would be the multi rule.

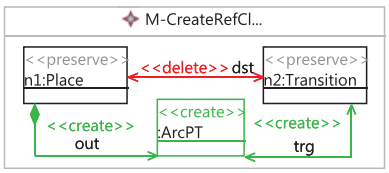

Fig. 7. Rule for replacing a connection by a connection class

In the third step we remove a direct reference from the meta-model. This rule corresponds to the *preserve* and *delete* parts of Fig. 6. In addition, the rule is equipped with a partial match by the references matched in the first step.

5 Tool Environment

Henshin [16] is developed in a joint effort of the Technische Univerität Marburg, the Technische Univerität Berlin and the CWI Amsterdam. The tool set is implemented in the context of the Eclipse Modeling Framework Technology (EMFT) [17] project, which in turn serves as an incubation project for the top-level project Eclipse Modeling. Henshin is currently comprised of three modules:

1. a tree-based and a graphical editor for defining transformation systems,
2. a runtime component, currently consisting of an interpreter engine, and
3. a state space generator and an extension point for analysis tools.

In the following, we briefly describe the state of the art of the Henshin tool set.

5.1 Editors

There are currently two editors available for defining model transformations in Henshin: i) a tree-based editor, generated by EMF itself, and extended with additional notation and tools to ease the editing of transformations, and ii) a graphical editor, implemented using GMF. Multi-panel editors, such as the one of EMF Tiger [6] and AGG [8], separate the editing of respectively left-hand side, right-hand side, and negative application conditions into multiple views. We chose an integrated view on transformation rules, similar to the Fujaba [18], GReAT [19], and GROOVE [20] editors. Examples of integrated transformation rules in the graphical editor are depicted in Figs. 4, 6 and 7.

5.2 Runtime

The Henshin runtime currently consists of an efficient interpreter engine. Given a transformation system and an EMF model as input, the transformation is performed directly, i.e., in-place, on the given model. For exogenous transformations, it further produces an additional output model instance. Note that endogenous transformations are particularly well-supported by the interpreter, since they are always executed in-place without the need of deep-copying model instances. The interpreter supports the full expressiveness nested conditions and transformation units, including amalgamations. Like EMF itself, the interpreter is independent of the Eclipse Platform and can be used in non-Eclipse applications as well.

5.3 Validation of Model Transformations

In Henshin, we currently provide the following validation support: To analyze in-place model transformations, we have developed a state space generation tool, which allows to simulate all possible executions of a transformation for a given input model, and to apply model checking, similar to the GROOVE [20] tool. Fig. 8 depicts the graphical state space explorer for the academic dining philosophers example. Here, the state space is finite, there exists one initial state (green, on the left) and two deadlock states (red, on the right), in which none of the

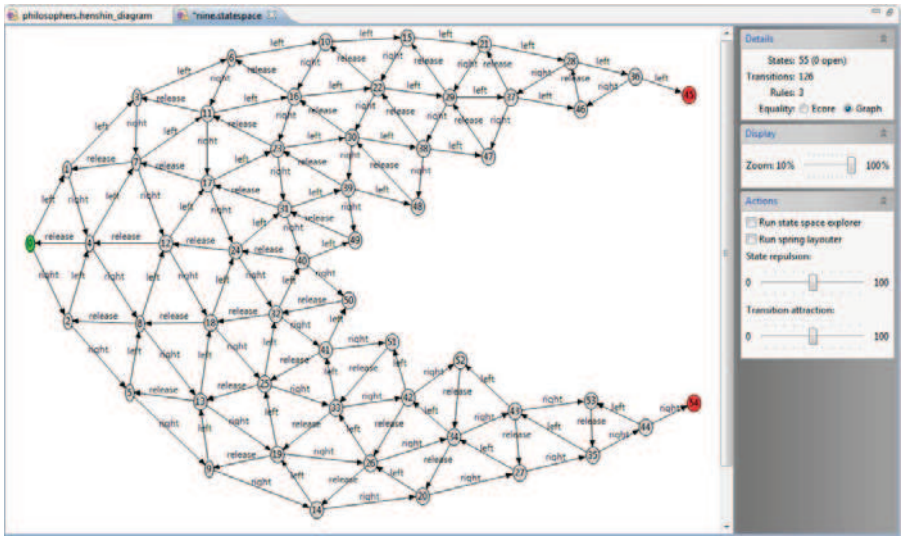

Fig. 8. State space generation tool

rules is applicable anymore. Large state spaces can also be generated and analyzed outside of the graphical tool. We use parallel algorithms for the state space exploration and can therefore benefit from modern multi-core processors. In its current version, our tool is able to handle state spaces with millions of states.

Our state space generator supports two different equalities for objects: i) the basic one defined by EMF itself (implemented in `EcoreUtil.equals()`), and ii) an equality based on graph isomorphisms. The latter abstracts from the order of elements in multi-valued references. In particular for highly symmetric models, such as the simple dining philosophers example, the use of graph equality reduces the size of the state space significantly. Note that Henshin currently does not provide means for recognizing the order of elements in multi-valued references. Therefore, the more compact state space induced by graph equality can be shown to be formally equivalent to the one generated using the basic EMF equality.

The state space tool set further provides an extension point for model checkers. We have integrated the third-party model checker CADP [21], which allows to verify temporal properties, specified as modal μ-calculus formulas. Moreover, we have integrated an existing OCL [22] validator for invariant checking. Found counter examples for both validation tools are shown as traces in the graphical state space explorer.

6 Related Work

Since model transformation is a key concept of model-driven development, a number of model transformation approaches have been developed. Especially two

kinds of model transformations are distinguished in MDD: (1) in-place model modification within the same language and (2) out-place translation of models to models of other languages or to code. Model transformation approaches supporting exogenous out-place transformations well are e.g. QVT, ATL, and Tefkat. We do not relate Henshin closer to these approaches due to space limitations. In the following, we consider EMF model transformations approaches for endogenous in-place transformations like Kermeta [4], EWL [5], Mola [23], Fujaba [18], EMF Tiger [6], and Moment2 [7] which we want to compare closer with Henshin.

Kermeta is an EMOF compliant textual approach to support behavior definition based on an action language which is imperative and object-oriented. Thus, Kermeta transformations are not rule-based and do not a formal foundation. Its tool environment includes a parser, a type-checker and an interpreter. The Epsilon Wizard Language is used to write small in-place transformations within the Epsilon project. The central concept are wizards which can be compared to rules. A wizard consists of a guard, a title and a do-section where the update is programmed in an imperative, object-oriented style. A formal foundation of the Epsilon Wizard Language is not mentioned. *EMF Tiger* is the predecessor of Henshin basing on graph transformation concepts as well. However, its transformation language is rather simple in the sense that it is purely rule-based and allows simple attribute changes only. Application conditions of rules are just sets of negative patterns. *Moment2* supports transformations of EMF models based on rewriting logic, as implemented in Maude. Its transformation language provides the concept of rewrites similar to graph transformation rules. Rewrites can be equipped with complex conditions expressed as OCL [22] constraints. However, rewrites cannot be composed to larger transformation modules. Due to its formalization based on rewrite logic, some static analysis and formal verification based on model checking are possible. *MOLA* supports transformations on EMF models where transformations are specified by MOLA diagrams consisting of graphical statements such as rules, loops, and calls to subprograms. An interpreter for Fujaba's story diagrams working on EMF models is presented in [18]. Both tools work directly on EMF models and offer similar language concepts as Henshin, namely rules based on patterns, and control constructs such as sequences and loops. However, a concept such as amalgamation is not offered by these tools. Furthermore, both MOLA and the story diagram interpreter do not have a formal basis for further validations of model transformations. *Viatra* [24] provides a rule and pattern-based transformation language combining graph transformation and abstract state machine (ASM) concepts. Modeling languages are defined by a proprietary meta modeling approach covering all main meta modeling concepts. The import of models in standard meta modeling formats such as EMF is supported as well. Based on graph patterns and rules, the Viatra transformation language offers advanced transformation features [24] including recursive graph patterns, generic and meta-transformations as well as control structures based on ASMs. Henshin's transformation features differ from

these especially concerning the execution of rules which might also be in parallel, as in amalgamated units.

Henshin is the only in-place transformation approach which comes along with a powerful transformation language being executed by a transformation engine that operates directly on EMF models. Moreover, its transformation features are all based on algebraic graph transformation [11,12,13].

Comparing Henshin's transformation language and tool set with the one of GROOVE [20], we can state that both are based on graph transformation and support nested application conditions as well as universal quantification using amalgamation. The use of regular expressions for matching is supported by GROOVE, but not by Henshin. Model checking in GROOVE is done using LTL or CTL formulas, whereas Henshin supports the more expressive modal μ-calculus through the CADP [21] model checker, as well as validation of OCL [22] invariants. To the best of our knowledge, GROOVE cannot handle EMF models yet.

7 Conclusion

In this paper, we present the Henshin transformation model for in-place transformations of EMF models. It builds up on graph transformation concepts such as rule-based transformation, nested and pattern-based application conditions for rules, and a variety of transformation units to define control structures for rule applications. To summarize, the Henshin transformation concepts rely basically on rules and patterns which can lead to a high amount of non-determinism when executing transformations. This amount can be reduced by the use of rule parameters, conditions and transformation units.

The direct execution of Henshin transformations allows a tight integration of transformations on inter-related EMF models as shown in the simple meta-model evolution example. Typing information can be dynamically loaded and re-loaded such that instance models can be re-typed over a modified meta-model. In the future, we intend to elaborate the translation of meta-model transformations to instance transformations further.

Although not addressed in this paper, Henshin can also be used for exogenous transformations such that source and target meta-model are integrated into correspondence meta-model in between. The transformation is formulated over this integrated meta-model. To view the target model only, we plan to extend Henshin by model operations such as projection operations restricting an instance model to the target domain.

References

1. EMF: Eclipse Modeling Framework, `http://www.eclipse.org/emf`
2. Steinberg, D., Budinsky, F., Patenostro, M., Merks, E.: EMF: Eclipse Modeling Framework, 2nd edn. Addison Wesley, Reading (2008)
3. MOF: Meta Object Facility (MOF) Core, `http://www.omg.org/spec/MOF`

4. Kermeta: http://www.kermeta.org
5. Kolovos, D.S., Paige, R.F., Polack, F., Rose, L.M.: Update transformations in the small with the Epsilon Wizard Language. Journal of Obj. Tech. 6(9), 53–69 (2007)
6. Biermann, E., Ehrig, K., Köhler, C., Kuhns, G., Taentzer, G., Weiss, E.: Graphical Definition of Rule-Based Transformation in the Eclipse Modeling Framework. In: Nierstrasz, O., Whittle, J., Harel, D., Reggio, G. (eds.) MoDELS 2006. LNCS, vol. 4199, pp. 425–439. Springer, Heidelberg (2006)
7. Boronat, A.: MOMENT: A Formal Framework for Model Management. PhD thesis, Universitat Politècnica de València (2007)
8. AGG: Attributed Graph Grammar System, http://tfs.cs.tu-berlin.de/agg
9. Biermann, E., Ehrig, K., Köhler, C., Kuhns, G., Taentzer, G., Weiss, E.: EMF Model Refactoring based on Graph Transformation Concepts. ECEASST 3 (2006), http://easst.org/eceasst
10. Cicchetti, A., Ruscio, D.D., Eramo, R., Pierantonio, A.: Automating co-evolution in model-driven engineering. In: 12th International IEEE Enterprise Distributed Object Computing Conference, pp. 222–231. IEEE Computer Society, Los Alamitos (2008)
11. Ehrig, H., Ehrig, K., Prange, U., Taentzer, G.: Fundamentals of Algebraic Graph Transformation. Monographs in Theoretical Computer Science. Springer, Heidelberg (2006)
12. Biermann, E., Ermel, C., Taentzer, G.: Lifting Parallel Graph Transformation Concepts to Model Transformation based on the Eclipse Modeling Framework. ECEASST 26 (2010), http://easst.org/eceasst
13. Kuske, S.: Transformation Units-A structuring Principle for Graph Transformation Systems. PhD thesis, University of Bremen (2000)
14. UML: Unified Modeling Language, http://www.uml.org
15. EMF Refactor, http://www.mathematik.uni-marburg.de/~swt/modref
16. Henshin, http://www.eclipse.org/modeling/emft/henshin
17. EMFT: Eclipse Modeling Framework Technology, http://www.eclipse.org/modeling/emft
18. Giese, H., Hildebrandt, S., Seibel, A.: Improved flexibility and scalability by interpreting story diagrams. ECEASST 18 (2009), http://easst.org/eceasst
19. GReAT: Graph Rewriting and Transformation, http://www.isis.vanderbilt.edu/tools/GReAT
20. Kastenberg, H., Rensink, A.: Model checking dynamic states in GROOVE. In: Valmari, A. (ed.) SPIN 2006. LNCS, vol. 3925, pp. 299–305. Springer, Heidelberg (2006)
21. Garavel, H., Mateescu, R., Lang, F., Serwe, W.: CADP 2006: A toolbox for the construction and analysis of distributed processes. In: Damm, W., Hermanns, H. (eds.) CAV 2007. LNCS, vol. 4590, pp. 158–163. Springer, Heidelberg (2007)
22. OCL: The Object Constraint Language, http://www.omg.org/technology/documents/formal/ocl.htm
23. MOLA: MOdel transformation LAnguage, http://mola.mi.lu.lv
24. Balogh, A., Varró, D.: Advanced model transformation language constructs in the VIATRA2 framework. In: SAC 2006: Proceedings of the 2006 ACM Symposium on Applied Computing, pp. 1280–1287. ACM, New York (2006), http://eclipse.org/gmt/VIATRA2